4

RADC-TR-90-47
Final Technical Report
April 1990

AD-A223 072

LANGUAGE SUPPORT FOR PARALLEL COMPUTATION

University of California

C.V. Ramamoorthy

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

RADC-TR-90-47 has been reviewed and is approved for publication.

APPROVED: [illegible]

PAUL M. ENGELHART
Project Engineer

APPROVED: [illegible]

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command and Control

FOR THE COMMANDER: [illegible]

IGOR G. PLONISCH
Directorate of Plans & Programs

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE April 1990	3. REPORT TYPE AND DATES COVERED Final Sep 88 - Sep 89

4. TITLE AND SUBTITLE

LANGUAGE SUPPORT FOR PARALLEL COMPUTATION

5. FUNDING NUMBERS

C - F30602-88-D-0027
PE - 63728F
PR - 2527
TA - 02
WU - P1

6. AUTHOR(S)

C. V. Ramamoorthy

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of California
Computer Science Division
Berkeley CA 94720

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Air Development Center (COEE)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RADC-TR-90-47

11. SUPPLEMENTARY NOTES

RADC Project Engineer: Paul M. Engelhart/COEE/(315) 330-4476

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This final technical report summarizes the research accomplished under the Expert Science and Engineering (ES&E) program by the University of California at Berkeley through Syracuse University. The research effort, entitled "Parallel Extensions for Object Oriented Programming", examined communication aspects leading to language support for parallel computation.

The work undertaken in this effort has extended concurrent programming languages in their communication primitives. Although many abstraction mechanisms have been used in programming languages, including control abstraction mechanisms (such as procedures) and data abstraction mechanisms (such as data types), communication abstraction mechanisms have been found to be the most useful for parallel/concurrent programming paradigms. Work accomplished in this effort has extended those mechanisms, concentrating on inter-process communication.

14. SUBJECT TERMS

Parallel Processing, Concurrent Processing, System Modeling, STOCS, Petri Nets

15. NUMBER OF PAGES

212

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 880922
Prescribed by ANSI Std. Z39 18
299-01

UNCLASSIFIED

13. ABSTRACT (Continued).

A formal model for concurrent systems, called the Synchronous Token-based Communicating State (STOCS) model, has been used to model and analyze concurrent systems. Since present concurrent languages do not support any form of analysis of the communication structure of programs, two new constructs based on STOCS formalism have been developed to support high level specification - handshake and unit. A fair and efficient algorithm for execution of multi-process shared events is also presented. [illegible]

UNCLASSIFIED

Table of Contents

CHAPTER 1

Introduction

The increasing use of computers in day-to-day life has placed demand on computing that is beyond the capabilities of current computer systems. This demand can be met either by increasing speed of uniprocessor systems or by increasing the number of processors in multi-processor systems. We call computer systems that use more than one processor, concurrent systems. The current hardware technology favors concurrent systems by making it more economical to provide high MIPS (million of instructions per second) by multiple processors rather than uniprocessors. In this dissertation, we will restrict ourselves to concurrent systems.

A concurrent system that consists of processors which execute in a lock-step manner is called a synchronous system. A concurrent system in which processors are loosely coupled and execute independently of each other is called an asynchronous system. Processors in an asynchronous system do not share the clock; therefore, it is easier to increase the number of processors in an asynchronous system than in a synchronous system. This dissertation deals only with asynchronous concurrent systems.

Asynchronous concurrent systems can further be classified into shared memory based and message based architectures. We call shared memory based systems, parallel. These systems assume that processors communicate with each other by writing and reading in shared memory locations. Concurrent systems that consist of multiple computers connected by a communication network are called distributed systems. Distributed systems offer many advantages over parallel systems. These advantages are as follows:

(1) Distributed systems provide *load sharing* to better exploit available processing capacity.

(2) Distributed systems provide *resource sharing.*

(3) Distributed systems provide *data sharing* as in distributed databases.

(4) The *geographical structure* may be inherently distributed. The low communication bandwidth may force local processing.

(5) The *logical structure* may be simpler, e.g. if each local process is located in a separate processor.

(6) The *reliability* of a system can be enhanced. Distributed systems are more reliable because the failure of a single computer does not affect the availability of others.

(7) The *flexibility* of a system is increased because a single processor can be added or deleted easily.

(8) *Availability of high bandwidth network and cheap diskless workstations* also favors distributed computing for economic reasons.

This dissertation deals only with message based concurrent systems. By concurrent systems we would mean distributed systems unless otherwise specified. Many of the techniques developed, however, will also be useful for parallel systems.

The usefulness of distributed systems has spurred a significant amount of research [Lampson 81, Alford 85, Raynal 88]. There have been advances both

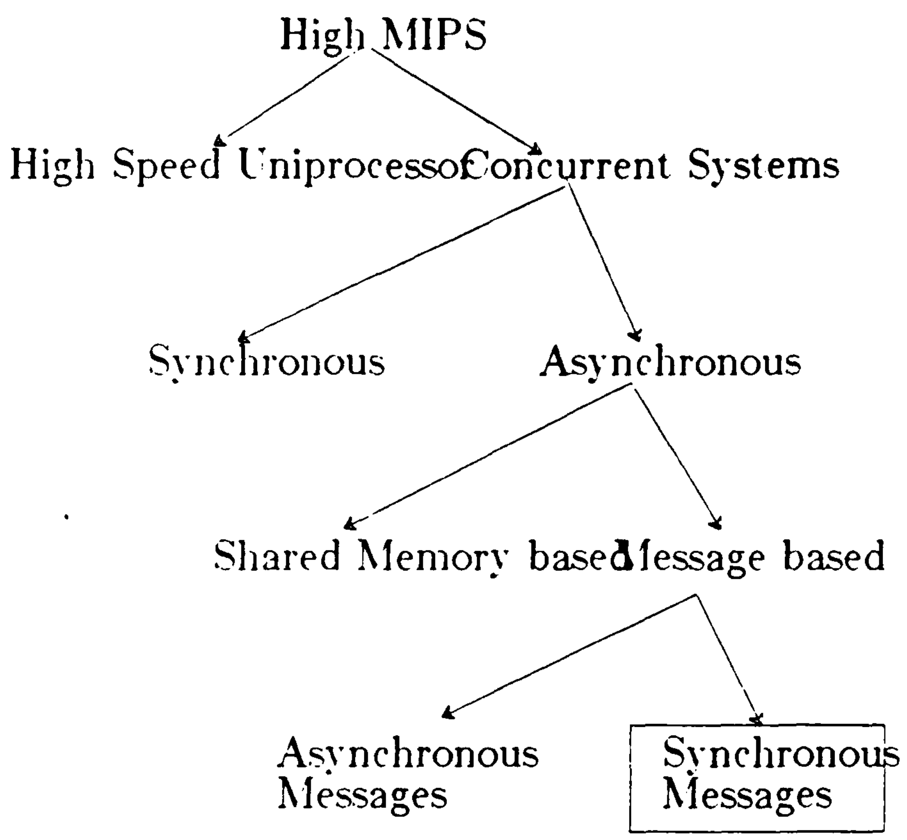


Figure 1.1: The Focus of this Dissertation

in hardware and software but the design of distributed software has proven to be more difficult than that of distributed hardware. Architectures, such as Hypercube, provide up to 16K processors connected by a network. The exploitation of such hardware still remains a challenging task. This dissertation deals only with techniques for the design of distributed software though many techniques developed will also be useful for designing distributed hardware.

The design of sound distributed systems requires formal specification and analysis techniques. Many algorithms informally argued to be correct, reveal errors in later analysis. Formal methods would eliminate this problem by avoiding any ambiguity that arises in informal reasoning. Formal specification also lends itself to automatic analysis. Figure 1.1 shows the focus of the dissertation.

1. Issues in Modeling Concurrent Systems

A model for a concurrent system would have different characteristics from models used in a different domain such as security. For example, we would expect a model for a concurrent system to have features for expressing communication and synchronization among multiple entities. In this section, we summarize the issues in formal specification and analysis of concurrent systems. These are as follows:

1.1. Processes: *Explicit vs Implicit*

As defined earlier, a concurrent system has more than one process. Some models assume that processes are specified explicitly by the user. Thus, the user is responsible for specifying what should be done by who. Alternatively, the model may be based on the idea of implicit parallelism. Here the user just specifies what needs to be done and the system arranges for its concurrent computation. The detection of parallelism has been found to be an extremely hard problem and therefore we have chosen to use explicit specification of concurrency in our model.

1.2. Communication: *Shared Variable vs Messages*

If multiple processes in a concurrent system need to coordinate, they must communicate with each other. This communication is traditionally expressed via shared variable model or explicit messages. The shared variable model assumes that a process can write into a shared memory location, from which other processes can read. In this paradigm, data structures are shared and synchronization is required to ensure that accesses and updates to the data is proper. Messages is an alternative form of communication in which processes do not share any data. A process has to explicitly send the information required by some other process. Lynch and Fischer [Filman 84] describe some of the difficulties of shared variable communication:

The way in which processes communicate with other processes and with their environments is by means of variables...Unlike message-based communication mechanisms, there is no guarantee that anyone will ever read the value, nor is there any primitive mechanism to inform the writer that the value has been read. Thus for meaningful communication to take place, both parties must adhere to previously-agreed-upon-protocols ..)

With the above in mind, we have assumed that communication is via messages in our model.

1.3. Buffering: *Unbuffered vs Unbounded Buffering*

An issue that is important for message based systems is the number of messages that can be pending at any time. Some systems provide *unbuffered* messages or *synchronous* messages which must be received by the receiver before the sender of the message can proceed. Other systems provide buffered messages or asynchronous messages which do not block the sender. Note that asynchronous systems can use either synchronous or aynchronous message passing. We have assumed synchronous message passing in our model. Hoare gives the following reasons for synchronized communication. (1) Synchronized communication is more basic as it matches closely a physical realization on wires which connect processing agents. Such wires cannot store messages. (2)

It also matches closely the effect of calling and returning from subroutines within a single processor, copying the values of the parameters and the results. (3) When buffering is desired, it can be implemented simply as a process; and the degree of buffering can be precisely controlled by the programmer. (4) The mathematical treatment of systems with unbounded buffering is also complicated by the fact that that every network is an infinite state machine even when the component processes are finite. (5) Buffering also makes fault recovery difficult as the failure of a **send** is detected much later in the program.

1.4. Expressive Power: *Richness vs Tractability*

The model should be rich enough to express important aspects of the system. For example, a finite state machine cannot model any system which can have an unbounded number of states, and may therefore be unsuitable to model some real life applications. A model that is theoretically more powerful, however, is inherently more difficult to analyze. For example, Turing machines, the most general model of computation known, are unanalyzable for most interesting problems, such as halting and equality. Any model for concurrent system should strike a good compromise between its expressive power and its analyzability. Our model is equivalent to Petri nets which are not only rich enough to capture unboundedness but also simple enough to guarantee that the halting problem is decidable.

Note that, even if two models have the same expressive power, they may have different expressive convenience. For example, Ada and Turing machine have the same theoretical power but it is much more difficult to write programs in Turing machine formalism than in Ada. Models that are used for proving properties about a system tend to be stripped of syntactic conveniences, while models that are used as implementation tools have syntactic sugar added to them.

2. Contributions of this Report

- A formal model for concurrent systems called the Synchronous Token based Communicating State(STOCS) is proposed. To prove that a STOCS machine is amenable to net-theoretic analysis, we prove that the reachability problem in a Petri net is reducible to that in a STOCS machine and vice-versa. The STOCS model is easier to use than Petri nets, as it supports modularity in specification and analysis. For example, we show that analysis of safety properties can avoid searching global state space, by considering only the relevant modules.
- We show that the STOCS model can be characterized algebraically by concurrent regular expressions. Concurrent regular expressions extend classical regular expressions with three operators - interleaving, interleaving closure and synchronous composition. Thus, the STOCS model combines advantages of both algebraic and net-theoretic approaches.
- We provide a unified treatment of *oracular* and *demonic* non-determinism in the STOCS model. We provide denotational semantics of STOCS machines and concurrent regular expressions which can take internal actions.

- Conventional automatic analysis techniques to catch logical errors in a concurrent system may be infeasible because the system may have a large, or even an unknown number of processes. These techniques, which are based on state space exploration, run into the state explosion problem. Since most distributed systems have one or more sets of identical processes, we exploit the symmetry to reduce the state space for automatic analysis techniques. We describe symbolic and inductive techniques to analyze a STOCS machine.

- Present concurrent languages do not support any form of analysis of the communication structure of programs. To support high level specification and analysis of distributed systems, we propose two new constructs based on STOCS formalism - *handshake* and *unit*. The handshake construct is a remote procedure call generalized for multiple parties. The unit construct has three functions - to restrict the possible calls to various handshake procedures, to provide a synchronization mechanism, and to specify computation that is directly relevant to communication. These constructs can easily be added to any existing language. The current system called ConC(Concurrent C) extends "C" for concurrent programming. A prototype of ConC runs on a Sun cluster operating under Unix 4.2 BSD.

- Implementation of a system expressed in the STOCS model requires execution of multi-process shared events. We present a fair and efficient algorithm for execution of multi-process shared events. We also present its application to distributed implementation of a generalized CSP alternative command. We show that our solution is superior to proposed implementations for generalized CSP alternative command.

3. Outline of the Report

Chapter 2 presents an overview of the previous work in modeling and analysis of concurrent systems. Chapter 3 provides the definition of the STOCS model. It also presents many modeling techniques for the STOCS model. Chapter 4 describes an algebraic characterization of a STOCS machine using concurrent regular expressions. Chapter 5 compares the STOCS Model with Petri nets. It shows by a constructive proof that the reachability problem is equivalent for STOCS machines and Petri nets. It compares the modeling convenience in Petri nets and STOCS machines. Chapter 6 addresses the issue of modeling internal actions using STOCS machines and provides denotational semantics for processes modeled using concurrent regular expressions. Chapter 7 describes projection, symbolic and inductive techniques to analyze a STOCS machine. It demonstrates these techniques by analyzing several examples: 2-out-of-3 problem, readers writers problem, the dining philosophers problem and the mutual exclusion problem. Chapter 8 describes Concurrent C (ConC), a programming language that extends C with constructs from the STOCS model. Chapter 9 presents a fair algorithm to execute multi-process shared events.

Following is the list of acronyms used in this report.

Acronym	Meaning
STOCS	Synchronous Token Based Communicating State
FLSTOCS	Free Labeled STOCS
DSTOCS	Deterministic STOCS
USTOCS	Uncontrollable STOCS
FSM	Finite State Machine
PN	Petri Net
FLOPN	Free Labeled Ordinary PN
RE	Regular Expression
CRE	Concurrent Regular Expression
UCRE	Uncontrollable CRE
ConC	Concurrent C

CHAPTER 2

Previous Work

The importance of concurrent systems has resulted in extensive research on formal models for expressing concurrent systems. Models capture the essential features of the system. In this chapter, we summarize the important models that have been developed to express concurrent systems. We evaluate these models for their suitability in modeling concurrent systems.

1. Petri Nets

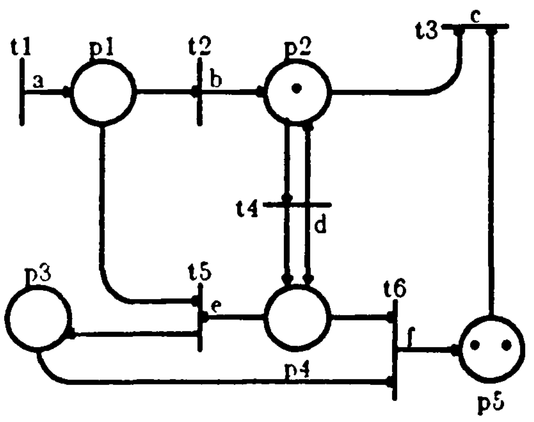


Figure 2.1 Petri Net example

Petri nets, first developed by Petri [Petri 62], have been immensely popular for specifying concurrent systems. A *general Petri net* (or simply a Petri net) is a directed bipartite multigraph. There are two kinds of nodes - *places* and *transitions* - represented by circles and lines, respectively. An example is shown in figure 2.1. It has five places(p_1-p_5) and six transitions(t_1-t_6). All arcs are drawn between a place and a transition or a transition and a place. There can be multiple arcs between the same pair of nodes, as seen between t_4 and

p_4. (A Petri net which has only simple arcs is called an *ordinary Petri net*. It is just a graph instead of a multigraph.) A marking of a Petri Net is a function which associates a certain finite non-negative number of "tokens" with each place of the Petri Net. In the Petri Net shown in figure 2.1, p_2 contains one token and p_5 contains 2 tokens. Arcs from places to a transition are called input arcs to the transition. Arcs from a transition to places are called output arcs of the transition. Transition $t5$ has two input arcs, one each from p_1 and p_4. Transition $t4$ has 3 output arcs, one to p_2 and two to p_4.

Therefore, a Petri net PN can be formally represented by a five-tuple (P, T, M_0, I, O), where:

- P is the set of places;
- T is the set of transitions;
- M_0 is the initial net marking;
- I, O: T --> P^W (the power set of P);
- I(t) is the set of the input places of transition t; and
- O(t) is the set of the output places of transition t.

A transition fires by removing tokens from the source places of its input arcs and puts tokens in the destination places of its output arcs. The number of tokens removed from a place when a transition fires is equal to the number of arcs from that place to the transition. Similarly, the number of tokens added to a place as a result of the firing of a transition is equal to the number of arcs from the transition to that place. The number of tokens in any place can never become negative, so a transition can fire only if there are sufficient number of tokens at the source places of all its input arcs. Such a transition is called "firable". In the Petri Net shown in figure 2.1, transitions $t4$ and $t3$ are firable. Transition $t3$ can fire by removing one token from p_2 and p_5. Since $t3$ has no output arcs, firing $t3$ does not add tokens to any place. If transition $t4$ fires, it adds 2 tokens to p_4. The number of tokens in p_2 remains 1, since $t4$ puts back the token it removes.

Each transition of a Petri Net can be associated with a label. (In the example transitions $t1-t6$ have labels a-f respectively.) A sequence of transition firings would be represented as a string of labels. We can also define an acceptance condition for the Petri Net. All configurations of the Petri Net satisfying the acceptance condition are final configurations. If a sequence of transition firings takes the Petri Net from its initial configuration to a final configuration, the string formed by the sequence of labels of the transitions is said to be accepted by the Petri Net. The set of all possible strings accepted by a Petri Net is called the language of the Petri Net. Different acceptance conditions and constraints on the labeling function yield different types of languages [Peterson 81]. [Peterson 81, Reisig 85, Genrich 80] provide an overview of research in the area of Petri nets.

2. Calculus of Communicating Systems (CCS)

The Calculus of Communicating Systems developed by Milner [Milner 80] had a profound impact on the science of specifying synchronous systems. The goal of his work is to develop a formal calculus for concurrent computation, similar to the way lambda calculus is a formal calculus of uniprocess computation. This formalism is based on two central ideas - synchronized communication and observation equivalence. Each concurrent system is described by means of algebraic expressions called behavior expressions. The calculus provides laws to prove the equivalence between two behavior expressions. Behavior expressions consist of multiple agents communicating by means of synchronous composition. A process is defined by the following syntax: P :: a.Q | Q+R | NIL | τ.Q | Q|R | Q | Q[S] The semantics of behavior expressions is (informally) as follows:

τ.Q: Process P acts as Q after a hidden action

a.Q: Process P acts as Q after the experiment

Q+R: Process P acts either as Q or R depending upon the choice offered

NIL: Process P does not admit of any experiment

Q|R: Process P act as composition of processes Q and R

Q....: Process P acts as Q with label *a* hidden

Q[S]: Process P acts as Q with the relabeling function S.

For example, a binary semaphore s, may be specified as s = PVs, which requires that a call to V must be made between two calls to P's.

CCS has been applied to provide semantics of programming languages such as CSP and NIL. Even though CCS provides an elegant formalism for understanding the meaning of concurrent systems, it is not useful as a specification tool for real systems. General CCS is Turing -equivalent and therefore unanalyzable for most properties.

3. Communicating Sequential Processes (CSP)

CSP, developed by Hoare[Hoare 85], provides a distributed language with a sound mathematical background. A CSP program is a static set of explicit processes. Pairs of processes communicate by naming each other in input and output statements. Thus, if A wishes to receive a value from B and store it in the variable x, it will execute the statement B ? x and this statement will block until process B executes an output statement by A!exp. Thus, communication is synchronous with unidirectional information flow.

Guarded commands are used to introduce indeterminacy. A guarded command is a conditional statement. The condition of the clause is a boolean expression. Optionally, it may have an input or output statement. For example, a process that merges characters from X, Y and Z and passes them to sink is specified in CSP as follows:

```
Merge:: c: character;
*[X?c -> Sink!c
  []
   Y?c -> Sink!c
  []
   Z?c -> Sink!c
]
```

The name of the process is Merge. * is the repetitive operator which executes a command repeatedly until all the guard clauses in the command fail. X?c is an input command. An input/output command fails if the process named in the command (X, in this case) has terminated. Sink!c is the action which is executed if the guard is true. [] provides indeterminacy and the process may choose any of the statements if more than one process is ready to communicate.

General CSP is Turing-equivalent and therefore cannot be analyzed automatically for most properties.

4. Actor Systems

The Actor model is based on the idea of object-oriented computation. This model was developed by Carl Hewitt and his colleagues at M.I.T. [Hewitt 79]. The discussion below is summarized from [Filman 84]. In an Actor system everything is an object called an actor. Actors communicate with each other by sending messages. Thus Actor system uses non-synchronized communication, in contrast to CCS. There are three kinds of actors: primitive actors, unserialized actors, and serialized actors. Primitive actors correspond to the data and procedure primitives of the computer system. For example, **2** and the function + are primitive actors. Serialized actors have a local state that the actor itself can change, while unserialized actors cannot change their local state. A typical unserialized actor is **factorial** which can be implemented in terms of other primitive and unserialized actors. A serialized actor associates state with a function. Serialized actors process messages serially - one at a time.

Every actor has a script (program) and acquaintances (data). When a message arrives at an actor, the actor's script is applied to that message. For example, an unserialized actor may accept messages like "add yourself to **3**, and send the answer to actor **G0042**".

The Actor metaphor provides uniform, independent entities that communicate with each other by message passing. Actor model thus provides a powerful formalism for expressing concurrent computation. No structure over waiting messages (e.g. ordering by send-time), and dynamic creation of actors poses difficulties for a tractable extensional theory of Actor Models [Milner 80].

5. Path Expressions

Path expressions were first defined by Campbell and Habermann [Campbell 74] as a synchronization mechanism. Using path expressions, a programmer can specify all constraints on the execution of operations. Code to enforce these constraints is generated by the compiler. The syntax of a path expression is:

path path_list **end**

A path_list contains operation names and path operators. Path operators include "," for concurrency, ";" for sequencing *n:(path_list)* to specify up to n concurrent activations of path_list, and "[path_list]" to specify an unbounded number of concurrent activations of path_list. For example,

path deposit, fetch **end**

places no constraints on the execution of deposit and fetch.

path deposit; fetch **end**

specifies that each fetch be preceded by an activation of a deposit. Synchronization constraints for a bounded buffer of size N are specified by

path N:(1:(deposit); 1:(fetch)) **end**

This mechanism was incorporated in Path Pascal. One of the main problems path expressions have is that it is difficult to include synchronization constraints that depend on the state of the resources.

Another formalism called COSY (Concurrent System) [Lauer 79] was inspired by path expressions. A path expression, referred to as a GR-path, is defined as follows: A GR-path is a string $P=P_1P_2..Pn$ where each P_i is an R-path. An R-path is a sequential constraint on the system expressed as a regular expression. Informally, if we think of P_i as describing the constraint c_i then the GR-path P describes a constraint c_1 and c_2 and $..c_n$.

It can be easily observed that since each R-path is a regular expression, a GR-path is just an intersection of regular expressions. In other words, a GR-path can model only a finite state system.

6. S/R Model

The S/R model is a state machine approach to specification and analysis [Aggarwal 87]. An S/R system consists of one of more processes. A process P is defined over a boolean algebra L as a five-tuple P = (V, S, σ, M, I) where:

- V is a set of states of P
- S is the set of selections of P, S $\subseteq$ L
- σ is the selector function of P, $\sigma: V \rightarrow 2^S$
- M is the transition matrix of P, $M: V \times V \rightarrow L$
- I is the initial state of P, $I \in V$.

The selector function associates with each state s the set of possible selections $\sigma(s)$ that can be made from state s. In our description, the selections will appear in curly brackets next to the state. The transition matrix is like an adjacency matrix of a directed graph with vertices V where the nonzero entities are actual labels describing the conditions for a transition to be enabled. Given an edge label $l=M(v,w)$ from state v to w, if the selection of a process is s in state v, then $s.l\neq 0$ means that the transition to w is possible. One can define a "calculus" of the processes so that the product of a process is again a process. Given processes $P_1,..,Pn$ with $P_i=(V_i,S_i,\sigma_i,M_i,I_i)$, the product of these processes is defined as $\mathrm{P} = \bigotimes_{i=1}^{n} P_i=(V,S,\sigma,M,I)$ where:

- $V=\overset{n}{\underset{i=1}{\times}} V_i$
- $S=\overset{n}{\underset{i=1}{p_i}}\, S_i$
- $\sigma=\overset{n}{\underset{i=1}{p_i}}\, \sigma_i$
- $I=\overset{n}{\underset{i=1}{\times}} I_i$

For example, consider two individual processes A and B that do not wish to be in the same room of a two room house consisting of an ATTIC and a CELLAR. A and B can choose to move from one room to the other (indicated by selections UP, DOWN) subject to the constraint that an individual in a room desiring to remain in the room has priority. The coordination between A and B can be shown as in Figure 2.2. In order to show that A and B are never both together, starting from the initial state A in ATTIC, B in CELLAR, the product of A and B is computed. The resulting process is shown in Figure 2.3 in which only the states (ATTIC, CELLAR) and (CELLAR, ATTIC) are reachable.

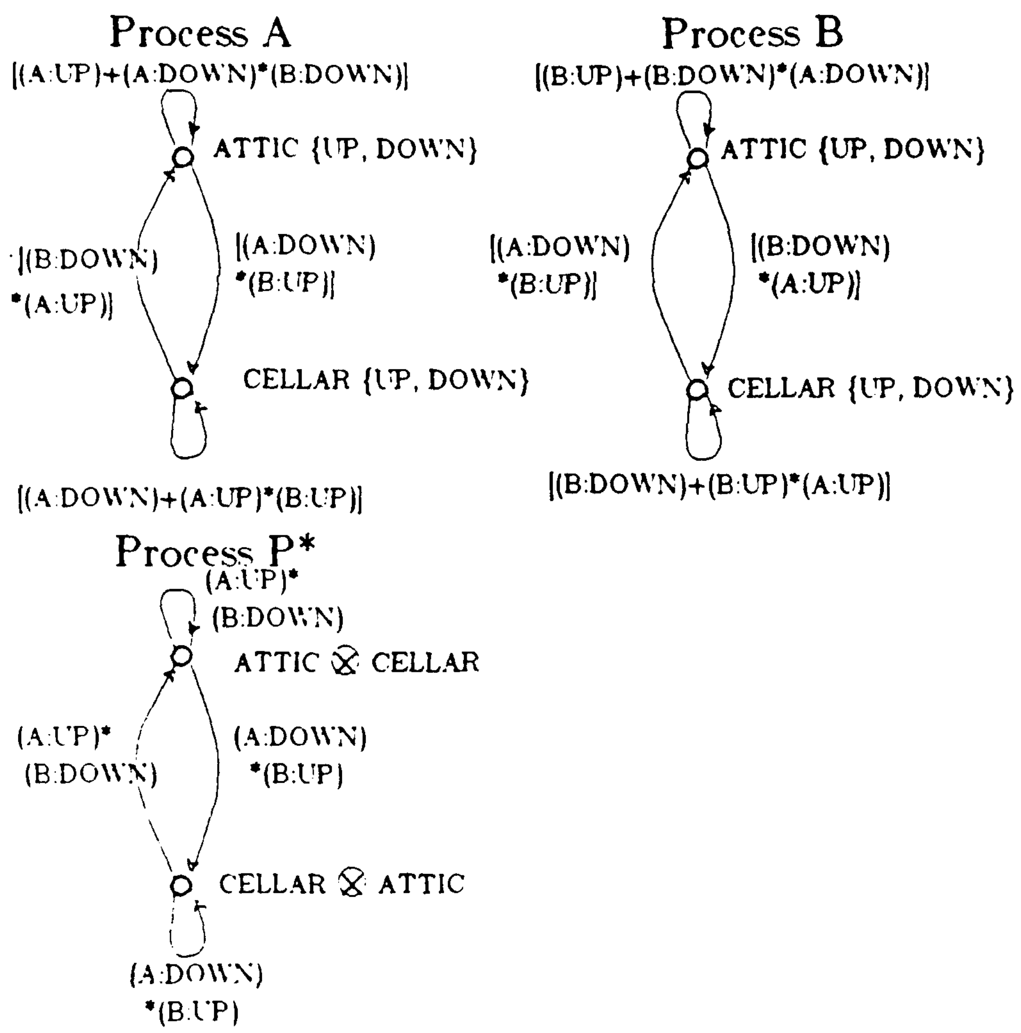


Figure 2.2: An example of S/R model

Since an S/R process has a finite number of states and a finite number of selections, an S/R process is theoretically equivalent to a finite state machine. The ability to label an edge with any boolean formula leads to a concise representation of many problems. This feature, however, also makes it difficult to provide an algebraic characterization of sequences of selections made by a S/R system. In addition, the system executes in two phases - selection and resolution. All processes make their local selections and then the global resolution is done. This paradigm may not be appropriate for specifying an asynchronous system with more than two processes.

7. Other Systems

Many other concurrent models have been developed for concurrent programming. Among Von Neuman languages are Ada [Ada 83], SR [Andrews 82], Occam [INMOS 84], PLITS [Feldman 79], Concurrent Pascal [Brinch Hansen 75], Concurrent C [Gehani 84] and Distributed Processes [Brinch Hansen 78]. In addition, there are data-flow languages [Ackerman 82] such as VAL, and applicative languges such as FP [Backus 78]. Since in this thesis we focus on automatically analyzable models, none of these models suits our purpose.

Many models have been proposed to study the semantics of concurrency and nondeterminacy. Some of these are Nivat's transition systems [Nivat 82], Winskel's event structures [Winskel 82], Pratt's pomsets [Pratt 82], Kahn's model [Kahn 77], Concurrent Transition Systems [Stark 87] and input/output automata [Steenstrup 83]. These models are useful for understanding semantics of concurrency, but their usefulness as specification tools remain to be seen.

8. Conclusions

All the above models have some good ideas and are suitable for some applications but the diversity shows us that there is no consensus about the *right* way of modeling concurrent systems. For implementation of concurrent systems, a model should have Turing-equivalent power so that all partial recursive functions are expressible. Thus we believe that Petri nets, finite state automata and derived models such as S/R model and Path expressions are inadequate for concurrent programming. Ada, CCS, and CSP are more suitable for concurrent programming with synchronous messages, whereas Actors and PLITS [Feldman 79] are more suitable for programming with asynchronous messages. Programs written in such languages can be proven correct only manually.

Before the actual implementation of the concurrent system, it is desirable to specify the crucial features of the system in some simple model which can be analyzed automatically. Petri net, COSY and S/R can be useful at this phase of software development. Petri nets have the advantage of being able to model unbounded number of states impossible to model in COSY and S/R. Petri nets, however, get very complex with an increase in the number of processes. This is because Petri nets do not support modularity for specifying concurrent systems with synchronous communication. COSY and S/R support modularity for synchronous communication and thus there is a need for a Petri net equivalent model with the modularity of COSY and S/R. The STOCS model, we believe, fills this need.

CHAPTER 3

The STOCS Model - Basic Definitions

1. Introduction

Concurrent languages such as Ada [Ada 83], CSP [Hoare 85] and Argus [Liskov 84] have good expressive power but any system that is specified using these languages can only be analyzed manually. As concurrent systems are difficult to design, the simplest of them can have subtle errors. To avoid these errors, we need to capture essential aspects of the system in a model and then analyze it for correctness. Models for concurrent systems that can be analyzed automatically have less expressive power than programming languages. They can be categorized roughly into two groups: *algebra based* and *transition based* models.

The algebra based models specify all possible traces of concurrent systems by means of algebraic operations on sets of traces. Examples of such models are path expressions [Lauer 75], behavior expressions [Milner 80] and extended regular expressions. Examples of tools to analyze the specifications based on such models are Path Pascal [Campbell 79], CCS [Milner 80] and Paisley [Zave 85]. Some of the commonly asked questions in such formal systems are: Is s a possible trace of the concurrent system under analysis? Is S_1, a concurrent system, the same as the concurrent system S_2?

The transition based models provide a computational model in which the behavior of the system is generally modeled as a configuration of an automaton. Examples of the *transition oriented* models are finite state machines

[Hopcroft 79], S/R Model [Aggarwal 87], UCLA graphs [Cerf 72], and Petri nets [Reisig 85]. Examples of modeling and analysis tools based on these models are Spanner [Aggarwal 87], Affirm [Gerhart 80] and PROTEAN [Billington 88].

In this chapter, we present a transition based model called the Synchronous TOken based Communicating State(STOCS) model. This chapter is organized as follows. Section 2 presents the STOCS model and many examples that can be modeled as STOCS machines. Section 3 describes a STOCS machine as a generalization of a finite state machine. Section 4 treats a STOCS machine as an acceptor of strings and describes its language. Section 5 describes deterministic STOCS machines which can be easily simulated to check for acceptance of a string. Section 6 describes the semantics of a STOCS machine by defining its language. Section 7 presents some paradigms for modeling by the STOCS model.

2. Synchronous Token based Communicating State(STOCS) Model

Informally, the STOCS model has five concepts - *unit, place, token, *-place* and *synchronous handshake*. A STOCS machine consists of one or more *units*. A *unit* is used to model a single process or a set of non-interacting processes. Each unit is an extended version of a finite state machine consisting of *places* and arcs between them. *Tokens* are used to model processes or data items. A **-place* models an unbounded number of processes or data items. *Synchronous handshake* is used for modeling interaction between processes. All executions in a STOCS machine take place in a synchronous manner.

Formally, a STOCS machine M is a set of units $(U_1, U_2, .., U_n)$ where each unit is a five-tuple, i.e. $U_i = (P_i, C_i, \Sigma_i, \delta_i, F_i)$ where:

- P_i is a finite set of *places*,
- C_i is an initial *configuration* which is a function from the set of places to natural numbers $\mathbb{N}$ and a special symbol '*', i.e. $C_i: P_i \rightarrow (\mathbb{N} \cup \{*\})$. This function represents the concept of *tokens* which may be thought of as residing in places. The symbol '*' represents an infinite number of tokens. The place which has * tokens is called a ***-place***. Other places are called *simple* places.
- Σ_i is a finite set of *handshake* labels
- $\delta_i \subseteq P_i \times \Sigma_i \cup \{\epsilon\} \times P_i$.
- F_i is a set of final places, $F_i \subseteq P_i$.

The configuration of a STOCS machine can change by the following *handshake rules (execution rules)*.

(1) A handshake with label a is said to be *enabled* if for all units $U_i = (P_i, C_i, \Sigma_i, \delta_i, F_i)$ such that $a \in \Sigma_i$ there exists a transition $(p_k, a, p_l) \in \delta_i$ with $C_i(p_k) \geq 1$. Informally, a handshake occurs simultaneously in all units which have that handshake in their handshake sets. Thus in example 3.1, Figure 3.1, *req* is enabled only if both p_2 and p_6 have tokens.

(2) A handshake a may take place if it is enabled. This will result in a new marking C'_i for all participating units, and is defined by

$C'_i(p_k) = C_i(p_k) - 1$

$C'_i(p_l)=C_i(p_l)+1$.

A *-place remains the same after addition or deletion of tokens. Figure 3.1 also shows the configuration of M after the execution of handshakes *pre* and *req*.

(3) If multiple handshakes are enabled, then any one of them can execute. For example in Figure 3.1(b) either *pre* or *crit* can fire. If a handshake is enabled in such a manner that it can fire in multiple ways, then the machine chooses the way in an oracular manner. This is analogous to a non-deterministic finite state machine accepting a symbol that is labeled on multiple out-going arcs.

Example 3.1 : Mutual Exclusion Problem

As an example of a STOCS machine, consider the mutual exclusion problem. where at most one process can execute the critical region. Each process does some pre-critical section processing, requests the permission to enter critical section, executes critical section, releases critical section, and then does some post-critical section processing. No two processes can be allowed to be in the critical section at the same time. A centralized solution can be expressed using a STOCS machine M as follows. M consists of two units, i.e. $M=(U_1,U_2)$ where:

$U_1=(P_1,\Sigma_1,C_1,\delta_1,F_1)$, $U_2=(P_2,\Sigma_2,C_2,\delta_2,F_2)$

$P_1=(p_1,p_2,p_3,p_4,p_5)$, $P_2=(p_6,p_7)$

$\Sigma_1=(pre,req,crit,rel,post)$, $\Sigma_2=(req,rel)$

$C'_1=(*,0,0,0,0)$, $C'_2=(1,0)$

$\delta_1=\{(p_1,pre,p_2),(p_2,req,p_1),(p_3,crit,p_4),(p_4,rel,p_5),(p_5,post,p_1)\}$

$\delta_2=\{(p_6,req,p_7),(p_7,rel,p_6)\}$

$F_1=\{p_1\},F_2=\{p_6,p_7\}$

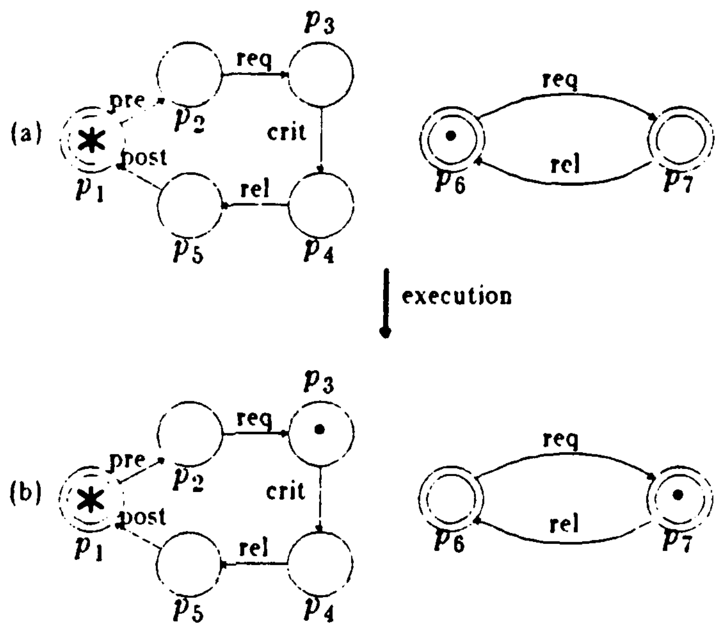


Figure 3.1: A STOCS machine for Mutual Exclusion

U_1 corresponds to any arbitrarily large number of processes that may be interested in executing the critical region. A process can execute *pre* handshake whenever it wants as *pre* does not require any coordination. Execution of *req*, however, requires participation of the coordinator process, which is possible only if the token is in p_6. U_2 corresponds to a critical region server which grants the permission needed to enter the critical region. A token in p_7 indicates that some process is executing the critical region.

The graphical representation of the STOCS machine for mutual exclusion is shown in Figure 3.1. Each place is represented by a circle, and each element of δ corresponds to a directed arc between a pair of circles. The initial configuration (alternatively called marking) is represented by placing the appropriate number of tokens in each circle. The number and position of tokens may change during *execution*.

Example 3.2: Producer Consumer Problem

This problem concerns shared data. The producer produces items which are kept in a buffer. The consumer takes these items from the buffer and consumes them. The solution requires that the consumer wait if no item exists in the buffer. A slight variant of this problem assumes that the buffer is bounded by n. The solution to both problems expressed in the STOCS model is given in Figure 3.2.

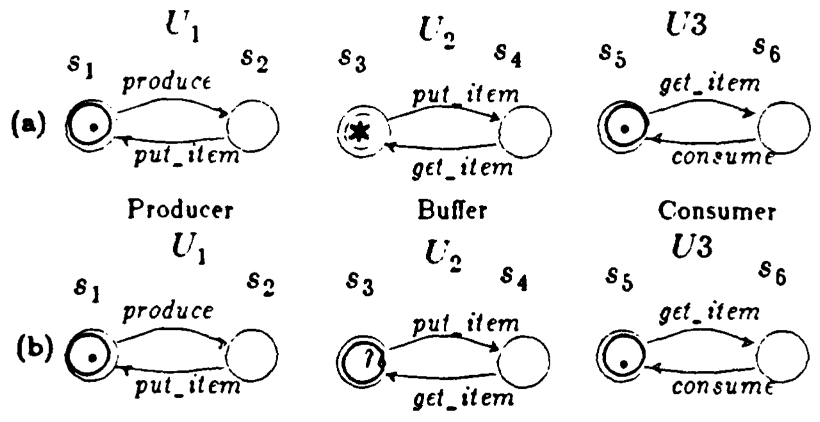


Figure 3.2: A STOCS Machine for Producer Consumer Problem

For both the problems, the consumer can execute *get_item* only if there is a token in the place p_4. If the buffer is unbounded, the producer never has to wait, whereas if the buffer is bounded, the producer may have to wait for the buffer to become empty (that is, for a token to be present at the place p_3 of the buffer). Thus, the number of tokens in the place p_3 represents the number of available buffers and the number of tokens in the place p_4 represents the number of filled buffers. Note how the *-place is used to represent an unbounded number of available buffers.

Example 3.3: 2-out-of-3 Memory Problem

The 2-out-of-3 problem is a good abstraction for many resource contention problems. Assume that a memory scheduler has three memory blocks and that any process requires two memory blocks to execute. The solution for a system with two processes is given in Figure 3.3. We place two token in the place s_1 to signify two processes and three tokens in the place s_5 to signify availability of three memory blocks. This example illustrates how multiple tokens can be used to represent multiple identical processes or multiple identical passive resources such as memory blocks.

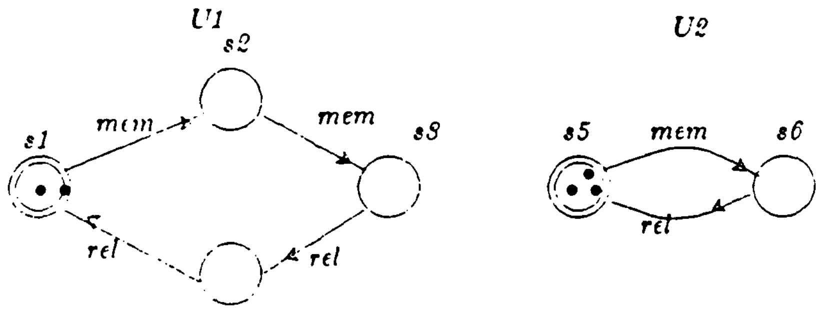


Figure 3.3: A STOCS Machine for 2-out-of-3 problem

3. Relationship of STOCS machines with Finite State Machines

In this section, we describe the STOCS machines in greater details by comparing its features with that of finite state machines.

(1) Current State vs Current Configuration

There is no concept of a *current state* in a STOCS machine, as there is in a finite state machine. *Tokens* are associated with places and can be thought of as residing in the places. A transition arc is said to be enabled when its tail has at least one token. A transition occurs by moving one token from its tail to its head. All tokens are identical and if there is more than one token in the tail, any one of the tokens may be moved. This has the advantage of ease in representing multiple resources, which may be active, such as processes, or passive, such as memory blocks. A *configuration* of a STOCS machine is a specification of the number of tokens in each place within each unit. The number of tokens in a place can be any finite positive number or a special symbol - *. The symbol * is used to represent an infinite number of tokens. If the number of tokens in a place is '*', it will always be '*' since we can add or subtract any finite number from infinity. These places are useful for modeling unbounded variables as in Example 3.2 and to model *forks* and *joins* as shown in Figure 3.4.

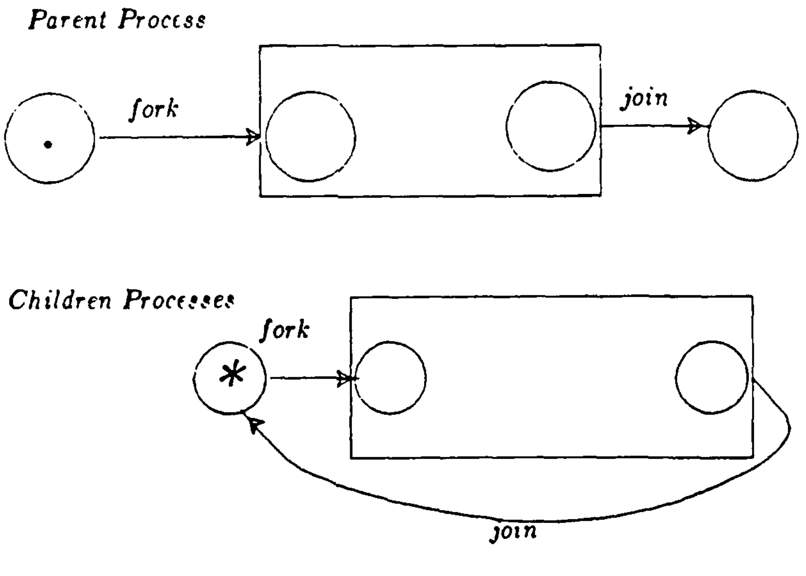


Figure 3.4: Modeling of *fork* and *join* in the STOCS Model

(2) Labeling of Transitions

As in finite state machines, each transition is given a label. All labels are derived from a finite set of symbols, Σ, the handshake set of the STOCS machine. If two or more units have transitions labeled with the same symbol, then these transitions must occur simultaneously, i.e. all these units must make the transition. This is the only form of interaction among the units.

(3) Power of a unit vs Power of a Finite State Machine

If a unit does not have any *-places then the number of tokens within the unit will be fixed since transitions only move tokens from one place to another. We can think of such a unit as a number of identical FSMs operating in parallel. Each FSM has the same set of states and transitions as the unit. Each token represents the current place of one FSM. When the unit makes a transition, moving one token from, say, place A to place B - one FSM with its current place A makes a transition to place B. Since all FSMs corresponding to a unit are identical, it does not matter which one of them makes the transition. A finite number of identical FSMs can be simulated by a single FSM· therefore, a unit with no *-places is no more powerful than an FSM since the number of states is finite. In fact, a unit with n places and m tokens can be converted into an FSM with no more than n^m states.

*-places give a unit more power. We can count up to arbitrarily large numbers using *-places. But within a unit, multiple *-places do not yield any additional power. If there is more than one *-place in a unit, we can merge all of them into one *-place. All transitions entering or leaving any of the original

*-places will now enter or leave the new *-place, because * represents infinity and no record can be kept of the number of tokens that have entered or left the state.

(4) Connectivity of the Graph

Note that unlike FSMs a unit need not necessarily be a single connected graph. If an FSM consists of more than one connected component, all components which do not contain the initial state can be deleted since these states are unreachable. In a unit we can have multiple components and each of these components can have tokens which move about within them. Movement of tokens within different components of the same unit is completely independent and need not synchronize with each other. However, tokens moving in different units must interact.

4. The Language of a STOCS Machine

In this section, we treat a STOCS machine as an acceptor of strings. We define the *language* of a STOCS machine and provide the motivation of its use. We also show that a class of STOCS machines called deterministic STOCS machines is particularly easy to use as acceptors of strings.

To use a STOCS machine as an acceptor of strings we need to define a certain initial configuration and a set of final configurations. A string is accepted if the STOCS machine starts from the initial configuration and arrives at one of the final configurations after making the transitions corresponding to the symbols in the string. (Obviously the string must be over the alphabet Σ). Instead of specifying the set of final configurations by ennumerating them we extend the definition of acceptance from finite state machines. We define a set of final places in each unit.

Definition: A configuration of a STOCS machine is a final configuration if there are no tokens in any non-final place.(i.e., all tokens in all units are in final places). By definition, all *-places are final places since the number of tokens in a *-place can never go to zero.†

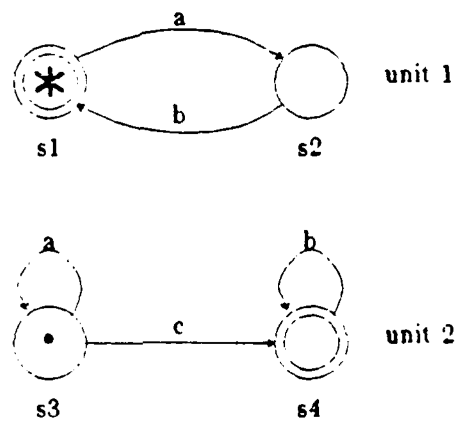


Figure 3.5: A STOCS machine accepting $a^n cb^n$

Consider the example shown in Figure 3.5. This STOCS machine consists of two units and its is alphabet $\Sigma = \{ a,b,c \}$. The first unit is formed by the places p_1 and p_2. p_1 is a *-place, and the other unit consists of places p_3 and p_4. Initially there is one token in place p_3. The final places of the system are p_1 (since it is a *-place) and p_4. Since symbols a and b are shared between the two units, transitions on a and b must be synchronized. The symbol c is not present in the first unit and can occur any time its transition is enabled in the second unit.

† If we allow *-places to be non-final places, then no string will be accepted by a STOCS machine.

This STOCS machine accepts the language $\{ a^n c b^n \mid n \geq 0 \}$. After a^n has been accepted, there will be n tokens in place p_2. The configuration of unit 2 will be identical to its initial configuration. On accepting c, the token in unit 2 will move to p_4. On each b, one token will be removed from p_2. When the number of b's becomes equal to the number of a's seen earlier, p_2 will become empty. This is an 'accept' configuration because places p_2 and p_4 will have no tokens. Note that unit 2 ensures that the string accepted is of the form $a^* c b^*$ while unit 1 ensures that $\#a = \#b$. One of the strengths of the STOCS model is its ability to use different units to model conceptually different properties of a language.

Definition: The *language* of a STOCS machine is defined as the set of all strings that are possible sequences of handshakes from the initial configurations to an accepting configuration.

For example, the language of the STOCS machine in Figure 3.6 is $\{a^n b^n \mid n \geq 0\}$. The motivation for the use of languages comes from the following:

1. Analysis: The language of a STOCS machine characterizes all sequences of actions that are possible in the system. A large class of interesting questions can be posed in language-theoretic terms. Some of these questions are: Is s a member of the language L, i.e. is the following string of computation feasible ? Is there a string that satisfies the temporal logic formula T? Is there a string which contains s as its substring ?

2. Characterization: The language of the STOCS model provide us with a

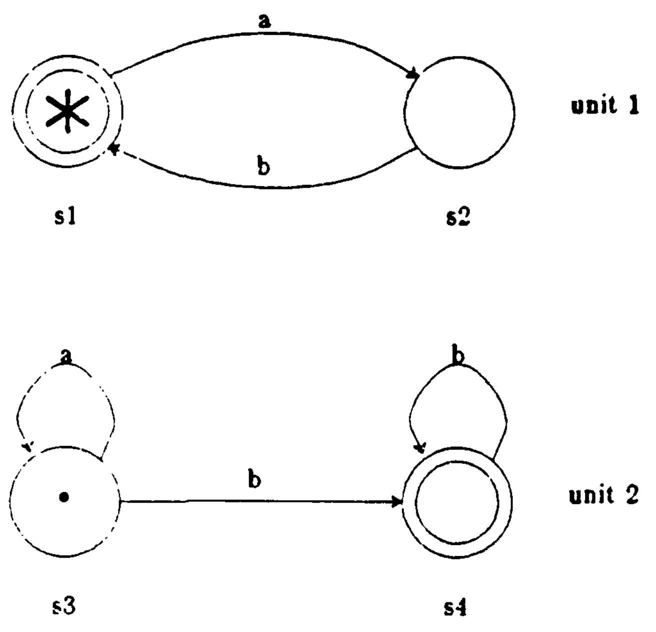


Figure 3.6: A STOCS machine accepting $a^n b^n$

way of specifying the behavior of a STOCS machine and we may chose to consider two machines equivalent if their languages are identical. Such a characterization gives us a chance to optimize a STOCS machine. Given a STOCS machine, one can reduce it to another STOCS machine by means of language preserving transformations. The new STOCS machine may be more desirable because it has less places, less units or more concurrency.

3. Synthesis of STOCS Machines: As we will see in Chapter 4, we can synthesize a STOCS machine given its specification in terms of concurrent regular expressions which are algebraic expressions on sets of strings. In general, given any form of specification of the language of a system, it is useful to generate a STOCS machine that accepts it.

5. Deterministic STOCS (DSTOCS) Machines

To check the acceptance of a string in a finite state machine, we simulate the machine on the string. If it is deterministic, we need only keep track of one *current* state during the simulation. If it is non-deterministic, we need to examine many possible paths or equivalently keep track of a set of possible current states. A sufficient condition for a finite state machine to be deterministic is that all arcs leaving a particular state have distinct labels and that there be no arcs labeled ϵ.

A STOCS machine is deterministic if, when simulating a string we need to keep track of only one "current configuration". This means that in any reachable configuration accepting a particular symbol leads to only one possible next configuration. With this motivation, we define a deterministic STOCS (DSTOCS) machine as a STOCS machine such that

(1) If a unit has exactly a single token then all arcs leaving a particular place have distinct labels and there are no arcs labeled ϵ. Such a unit is equivalent to a deterministic finite state machine.

(2) If a unit has multiple tokens, then it must be free-labeled. A unit is free-labeled if it has distinct labels on all of its arcs and there are no arcs labeled ϵ.

Simulation of a DSTOCS machine requires remembering only a single configuration because on a given symbol a unit with a single token can move to only one possible place and a unit with multiple tokens has only one arc labeled with it. The STOCS machine in Figure 3.6 is deterministic because unit 2 has a single token and it satisfies our condition for units with single token, while unit 1 is free labeled.

Note that units with single tokens are essentially finite state machines and are therefore good for putting regular constraints on the language. Such units are, however, not good for counting which is done by units which are free-labeled. A STOCS machine which consists of only free-labeled units is called a *Free Labeled* STOCS (FLSTOCS) machine. (terminology borrowed from Petri Net theory).

The STOCS machine in Figure 3.5 is a FLSTOCS machine because each arc in every unit has a unique label. The STOCS machine in Figure 3.6 is not an FLSTOCS machine because unit 2 has two arcs labeled b. FLSTOCS machines are good only for counting and may not even accept finite languages. We show that there can be no FLSTOCS machine accepting $P = \{\epsilon, a, ab, abb\}$.

To show this result, we need the following definition and Lemma.

Definition: A language P is a *prefix closed* language if for any s that belongs to P all prefixes of s also belong to P. Note that a prefix closed language must contain ϵ. Languages $\{\epsilon, a, aa, aaa, ..\}$ and $\{\epsilon, a, ab\}$ are prefix closed. Language $\{\epsilon. ab\}$ is not prefix closed because it does not contain the string a, a prefix of ab.

Lemma 3.1: For any prefix closed language P, if an FLSTOCS machine S accepts it then it is also accepted by a FLSTOCS machine S' with only final places.

Proof: Construct S' by deleting all non-final places in S. Clearly $L(S') \subseteq L(S)$, since a path that can be traced by tokens in S' can also be traced in S.

Also $L(S) \subseteq L(S')$. This is because any path that is traced for accepting a string in S cannot pass through a configuration in which a token is in a non-final place, otherwise the set will not be prefix closed. *Q.E.D.*

Theorem 3.1: There is no FLSTOCS machine accepting the language $P=\{\epsilon,a,b,ab,abb\}$.

Proof: Assume, if possible, there exists a FLSTOCS machine S that accepts P. Since P is a prefix-closed language, by Lemma 3.1, we can convert it to a machine which does not have any non-final places.

Since the string *ba* does not belong to the language, after a *b* has occurred in the input string, at least one arc labeled *a* should be disabled. The only way an occurrence of *b* can disable a transition labeled *a* is by removing all tokens in the source place of an arc labeled *a*. For this to happen there must be an arc labeled *b* leaving this place. This means that after the first *b* appears in the input string, the arc labeled *b* will also be disabled since there will be no tokens in its source place. Hence S will reject the string *abb*. *Q.E.D.*

From the above discussion, we note that a DSTOCS machine combines capabilities for checking that symbols are in proper sequence by means of regular sets, and for checking that symbols are in proper count by means of FLSTOCS machine. As an application of DSTOCS machines we show a DSTOCS machine in Figure 3.7 which accepts all valid arithmetic expressions. Unit 1 is a finite state machine which checks the sequence of all symbols without counting them. Unit 2 uses a *-places to count the number of parentheses.

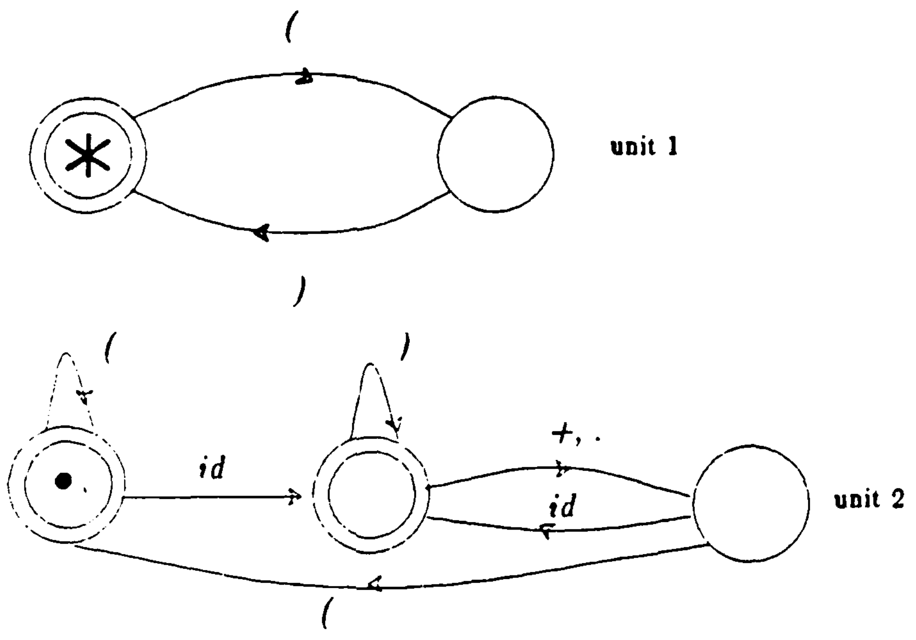


Figure 3.7: A DSTOCS machine to Parse Arithmetic Expressions

6. Semantics of the STOCS Model

In the following section, we provide an extensional theory of STOCS machines. Our theory of concurrent processes is built on following assumptions:

(1) **Non-simultaneity of Events**: We assume that two events cannot be observed simultaneously. If the simultaneity of a set of events is important (e.g. in synchronization), we represent the set of events as a single event occurrence. If the simultaneity is not important, we allow occurrences of events to be recorded in any order. Milner, who uses the same assumption in his proposal of CCS, justifies it by quantum theory which states that the flow of information is bounded by the speed of light and therefore if two events happen simultaneously, they will be recorded at different times by the observer. This assumption also makes the entire theory more elegant and tractable.

(2) **Atomicity of an Event**: In this theory we will ignore detailed timing consideration of events, and each event will be considered atomic in nature. Thus, no analysis can make an assumption about the time duration of events. A time-consuming action is represented by a pair of events, the first denoting its start and the second denoting its end. The interval between these events represents the duration of the event and it may overlap with other events.

(3) **Non-probabilistic Analysis**: We will not make any distinction between two systems that show the same possible behavior but each behavior with different probability. For example, a coin which on a toss shows head with probability 0.6 is considered equivalent to a coin which shows head with probability 0.5 but different from coins which show head with probability either 0 or 1.

(4) **Non-randomness in Execution**: We call a machine non-random if for any string, the machine either accepts or reject the string, but will always return the same answer. Thus the set of strings that are rejected is the exact complement of the set of strings that are accepted. STOCS machines that can return different answers for the same input at different times are called Uncontrollable STOCS (USTOCS) machines and are the subject of Chapter 6.

With above assumptions, we are ready to define equivalence of two STOCS Machines. We call two concurrent systems equivalent if an external observer cannot distinguish between the two systems no matter how different their internal structure. The observer (or environment) is allowed to give an input string to the machine and observe whether the machine accepts or rejects it. Since for deterministic processes, a string is always either accepted or rejected, all strings that do not belong to the acceptance set are always rejected. The external behavior of these processes, therefore, can be characterized as a tuple (Σ , L) where Σ represents the set of events that the process engages in and L is the language of the STOCS machine. Formally,

Definition: Two STOCS machine M_1 and M_2 are *equivalent* if their alphabet and language is the same.

The *alphabet* of a machine is the set of events a machine can possibly engage in. For example, the machine in example 3.1 can only engage in {*pre*, *req*, *crit*, *rcl*, *post*} and therefore cannot engage in event *put_item*. The following STOCS machines are considered equivalent. Both of them consist of a single unit as shown in the Figure 3.8. The behavior of both the machines can be characterized as $((a,b,c),(ab,ac))$. On the other hand, the machines shown in Figure 3.9 are different because their behaviors are $((a,b,c),())$and$((a,b),())$ respectively.

7. Modeling by the STOCS Model

In this section, we provide paradigms for modeling by the STOCS formalism.

7.1. Event and Conditions

A condition is modeled using place, and events are modeled using handshakes. An event that depends on conditions of multiple entities is shared

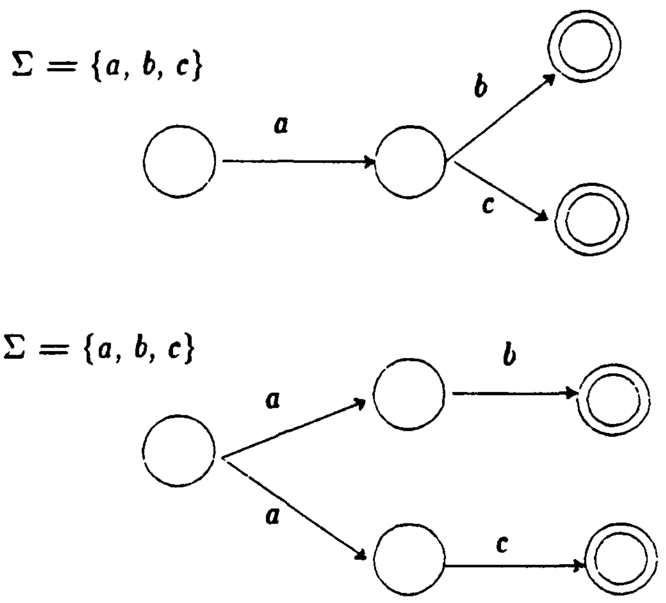


Figure 3.8: Equivalent STOCS Machines

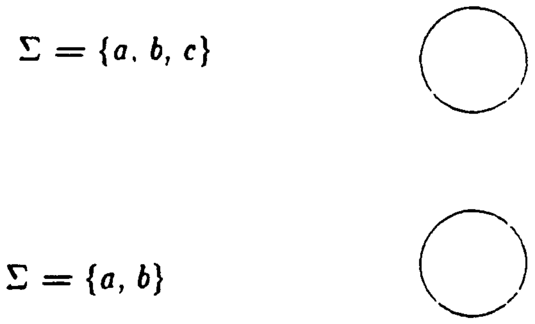


Figure 3.9: Different STOCS Machines

by multiple units. For example, consider the problem of modeling a simple machine shop. Various conditions and events for the system are as follows.

condition:

- s1: order arrived and waiting
- s2: order being processed
- s3: order complete
- s5: machine shop waiting
- s6: machine working on the order

events:

- e1: an order arrives
- e2: processing starts
- e3: processing completes
- e4: order delivered

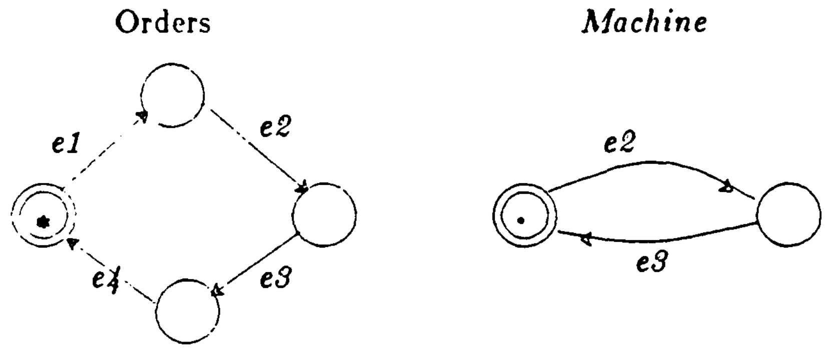


Figure 3.10: A STOCS machine for machine shop modeling

Figure 3.10 shows a STOCS machine for modeling the machine shop. Let us consider a more complex situation to bring out the advantages of modularity in the STOCS model. The machine shop may have three machines - M1, M2 and M3. It may have two operators F1 and F2. An order needs two stages of machining. First, they must be machined by M1 and then by either M2 or M3. F1 can operate M1 and M2 while F2 can operate M1 and M3. Figure 3.11 shows the modeling by a Petri net and Figure 3.12 shows its modeling by a STOCS Machine. In the STOCS machine the communication is hidden, each process is specified independently. This means that it is easier to understand and write specifications in the STOCS model. It is also easier to specify a partially developed system.

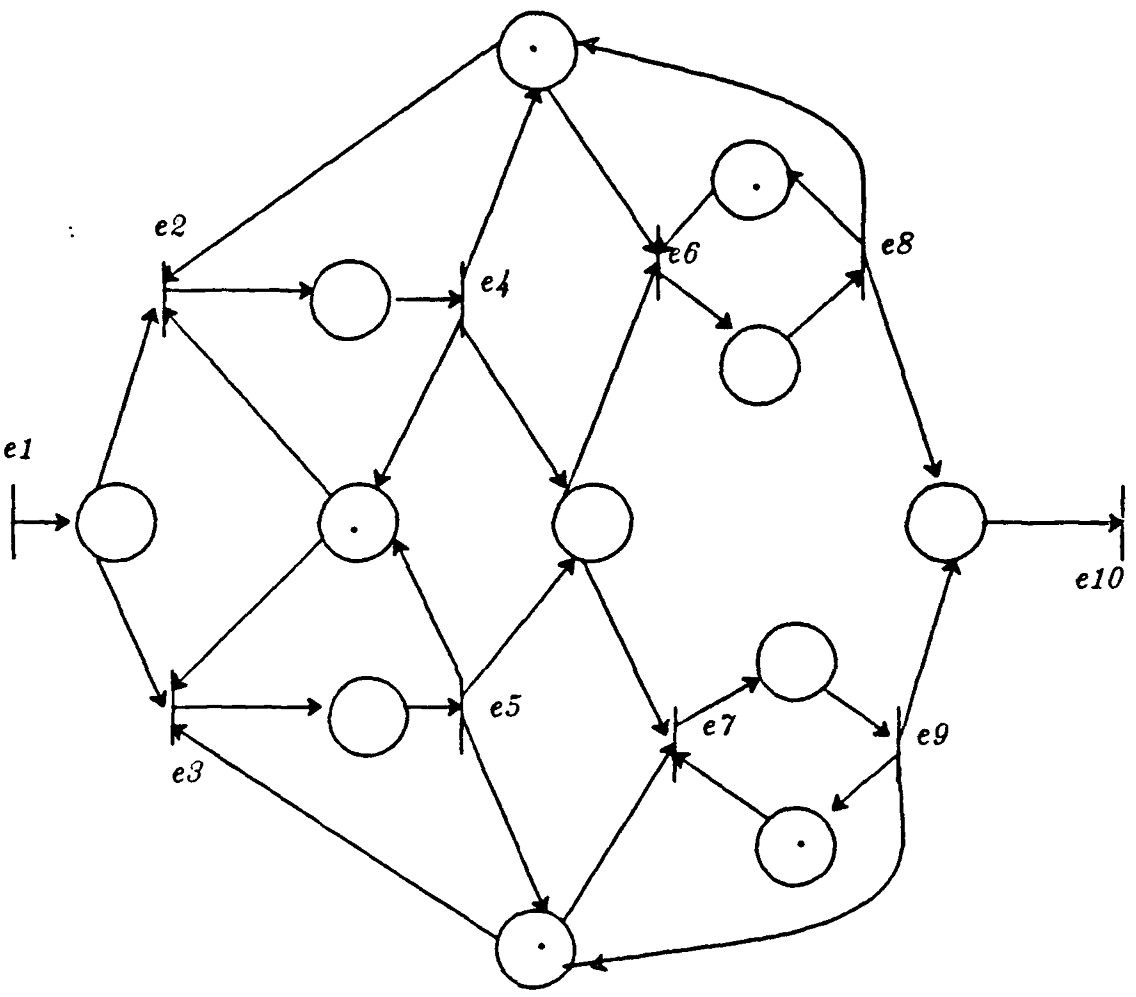


Figure 3.11: A Petri Net for Complex Shop modeling

7.2. Concurrency and Choice

The concurrency is present in the model as more than one transition can be enabled at one time. In Figure 3.13, *a* and *b* model concurrency as they can be fired in any order. The choice is modeled in the system by means of multiple arcs emanating from a single node. For example, in Figure 3.13 either *a* or *b* can fire but not both.

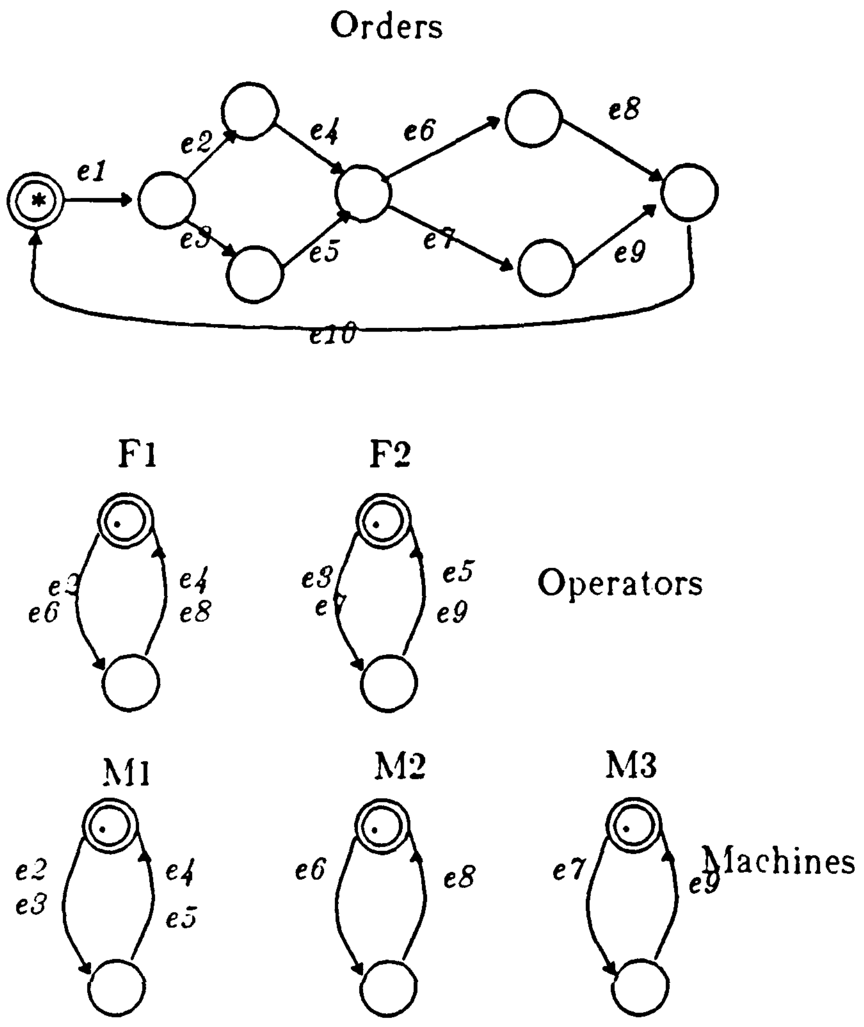


Figure 3.12: A STOCS Machine for Complex Shop modeling

7.3. Linear Constraints on the Language

We can also model systems specified by constraints posed on their event sequences. One of the main weaknesses of the finite state model was its inability to count an arbitrary number of instances of an event. Thus it cannot accept languages L = $\{a^n.b^n \mid a,b\epsilon\Sigma, n \geq 0\}$. The language L can be written as the conjunction of two constraints:

(1) All a's precede all b's.

(2) the number of a's = the number of b's

The first constraint can be checked by a finite state machine corresponding to

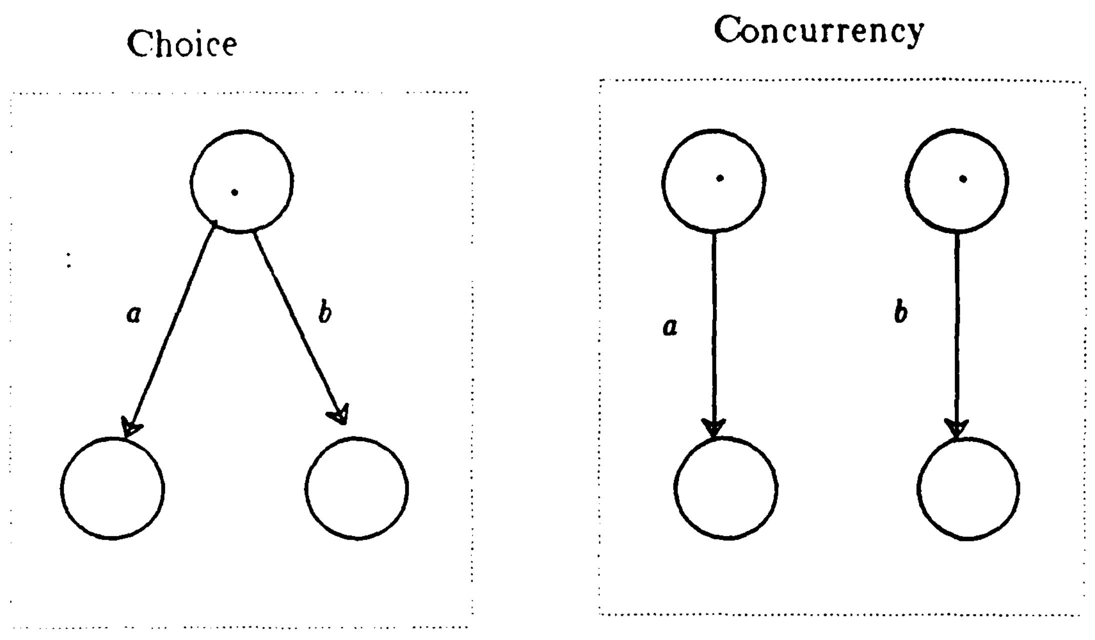


Figure 3.13: STOCS Machine representing Choice and Concurrency

the regular expression a^*b^*, but the second constraint can not be checked by a finite state machine due to the pumping lemma. If N_a represents the number of a's in the string and N_b represents the number of b's in the string, the second constraint can be written as $n_a = n_b$. Figure 3.7 shows a STOCS machine for L with two units, one for each constraint. To illustrate the modeling power of STOCS machines, we also show the modeling of following constraints in Figure 3.14.

(1) $n_a + n_b = n_c$

(2) $n_a = 2n_b$

7.4. Interaction Between Multiple Systems

It is easy to capture the interaction between multiple systems by means of shared handshakes and the definition of synchronous execution. Thus, if M_1 and M_2 are two STOCS machines consisting of $(U_1, U_2 .., U_m)$ and

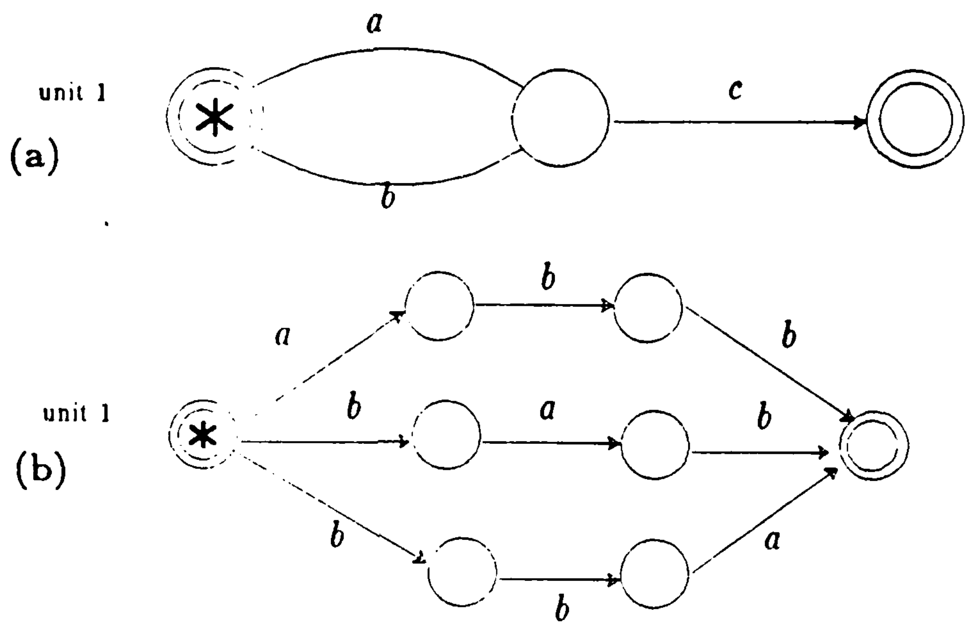


Figure 3.14: STOCS Machines to model integral linear constraints

$(U_1', U_2', \ldots, U_n')$, then the STOCS machine resulting from their interaction is simply $(U_1, .. U_m, U_1', .. U_n')$. The interaction between these STOCS machines follows from the definition of synchronous execution. A synchronous handshake requires that all units with that particular handshake in their handshake set participate. Therefore, a handshake which could have taken place before composition with another STOCS machine may have to wait for units in the other STOCS machine to synchronize.

For example, consider a chocolate vending machine. There are two kinds of events: *choc*, which is the dispensing of a chocolate, and *coin*, which is the depositing of a coin by the customer. The machine owner specifies that the number of *choc* events should be less than or equal to the number of *coin* events (Figure 3.15). The customer, on the other hand, will not deposit a coin until he receives the chocolate for his last coin (Figure 3.15). Hence, when these two machines interact, the only feasible sequence of events is *coin choc* *coin choc etc.*

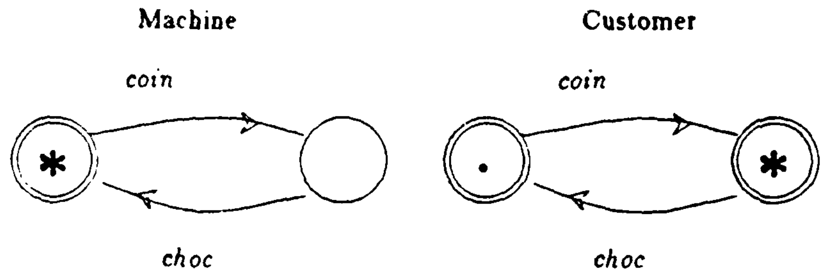


Figure 3.15: A STOCS machine for a Chocolate Vending Machine and a Customer

8. Conclusions

In this chapter, we have defined a transition based model called the Synchronous Token based Communicating State (STOCS) model. A STOCS machine consists of units, each of which models a set of non-interacting processes. We have presented many examples modeled by STOCS machines. STOCS machines can easily model concurrency and synchronization making them useful for specifying concurrent systems. We have shown how STOCS machines can act as acceptors of strings. We have, thus, defined the language of a STOCS machine. Based on the language and the alphabet of a STOCS machine, we have defined the equivalence between two STOCS machines.

CHAPTER 4

An Algebraic Characterization of STOCS

1. Introduction

In Chapter 3, we describe a transition based model to define a process. An alternative approach to specifying a concurrent system is based on algebra. In this approach, a system is built by applying algebraic operators on sub-systems in a well defined manner. An *equivalence* of a transition based model and an algebraic model provides us the flexibility of specifying a system in an algebraic model and analyzing it in automaton model and vice-versa. For example, finite state machines and their equivalent algebraic expressions (regular expressions) serve as an excellent vehicle for specification of sequential systems. A finite state machine has an *equivalent* regular expression, means that the language characterized by them is identical. Many tools, such as LEX, take advantage of this equivalence to convert specifications expressed in algebra based models to transition based models for lexical analysis of programming languages. In this chapter, we define concurrent regular expressions and show that they are equivalent to General STOCS. Figure 4.1 summarizes the relationships between various transition based and algebraic models.

We shall use the following naming conventions. Words in lower case denote distinct events, e.g. *get, put, a, b*. The letters A,B,C stand for either the concurrent regular expressions or the language of a process characterized by them.

This chapter is organized as follows. Section 2 defines concurrent regular expressions. Section 3 shows examples of concurrent systems modeled by concurrent regular expressions. Section 4 establishes their equivalence with STOCS machines. Section 5 describes the languages characterized by concurrent regular expressions.

$$\begin{tikzcd}
\text{finite state machines} \arrow[r] \arrow[d, leftrightarrow] & \text{units} \arrow[r] \arrow[d, leftrightarrow] & \text{STOCS Machines} \arrow[d, leftrightarrow] \\
\text{regular expressions} \arrow[r] & \text{unit expressions} \arrow[r] & \text{Concurrent regular expressions}
\end{tikzcd}$$

Figure 4.1: Relationship between Various Automata and Algebraic Expressions

2. Concurrent Regular Expressions

To motivate the definition of concurrent regular expressions, we note that regular expressions specify the computation of essentially a sequential finite state machine, and are unsuitable for expressing the languages of the concurrent machines. To specify the trace of a concurrent system, we have proposed an extension of regular expressions (r.e.) called concurrent regular expressions (c.r.e.). Recall that an r.e. over an alphabet Σ is defined as follows:

1) Any a that belongs to Σ is an r.e. defined over {a}.

2) If A and B are r.e.'s defined over Σ_A and Σ_B, then A.B (concatenation) and A+B (or) are r.e.'s defined over $\Sigma_A \cup \Sigma_B$, and A* (Kleene closure) is an r.e. defined over Σ.

For example, if $\Sigma = \{a,b\}$ then $a^*b+b^*a, abb, ab+ba$ are some examples of regular expressions defined over Σ. To define concurrent regular expressions we add the following operations: $||$, α, $[]$. With these additional operators we define a concurrent regular expressions (c.r.e.) over an alphabet Σ. A c.r.e. **A** is characterized by two sets - its alphabet set (Σ_A), and its language (L_A). Even though a concurrent regular expression is defined over Σ, we will not explicitly use Σ in defining concurrent regular expressions. Any binary operator over two different alphabet set results in a concurrent regular expression defined over the union of two alphabet sets. Thus the expression $A\ op\ B$ is always defined over $\Sigma_A \cup \Sigma_B$. As a result, we will also treat in this chapter a c.r.e A synonymous to its language L_A.

2.1. Definition

(1) Any a that belongs to Σ is a regular expression (r.e.) defined over {a}. A special symbol called ϵ is also a regular expression defined over {}. If A and B are r.e.'s, then so are A.B (concatenation), A+B (or), A* (Kleene closure).

(2) A regular expression is also a *unit* expression. If A and B are unit expressions then so are A^α, B^α, and $A||B$.

(3) Any unit expression is also a *concurrent regular expression*. If A and B are concurrent regular expressions then so is $A[]B$,

Examples of some valid concurrent regular expressions are $(a^*b)^\alpha||b^*c[](ab)^\alpha$ and $(ab)^\alpha||(ba)^\alpha$. Some invalid concurrent regular expressions are $(ab)^\alpha(bc)^\alpha$ and $((ab)^*[](ac)^*)^\alpha$. These expressions are not valid because they use the α operator in a manner not permitted by the syntax. Table 4.1 summarizes semantics of all the operators described by an example.

Operator	Name	Result
A+B	Choice	$\{ab, ba\}$
A.B	Concatenation	$\{abba\}$
A*	Kleene Closure	$\{\epsilon, ab, abab, ..\}$
A || B	Interleaving	$\{abba, abab, baab, baba\}$
A^{α} :	Alpha Closure	$\{\epsilon, ab, abab, aabb, ..\}$
A[]B	Composition	$\{\}$

Table 4.1: Example for A = $\{ab\}$ and B = $\{ba\}$

2.2. Choice, Concatenation and Kleene Closure

These are the usual regular expression operators.

Choice between two set of strings is defined as follows.

$A+B=A\cup B$

For e.g. if A = {ab,bc} and B = {a,c} then A+B = {ab,bc,a,c}.

Concatenation of two sets of strings is defined as follows.

A.B = $\{w/w=s.t \text{ where } s\in A \wedge t\in B\}$.

Kleene closure of a set A is defined as

$$A^*= \bigcup_{i=0,1,\ldots} A^i$$

Properties of these operators are as follows:

1) A+A = A
2) A+B = B+A
3) A+(B+C) = (A+B)+C
4) A+ϕ = A
5) A.(B.C) = (A.B).C
6) A.$\{\epsilon\}$ = A
7) A.(B+C) = A.B + A.C
8) $(A^*)^* = A^*$

For details of these operators the reader is referred to [Hopcroft 79].

2.3. Interleaving

To define concurrent operations, it is especially useful to be able to specify the interleaving of two sequences. Consider for example the behavior of two independent vending machines VM1 and VM2. The behavior of VM1 may be defined as (coin.choc)* and the behavior of VM2 as (coin.coffee)*. Then the behavior of the entire system would be interleaving of VM1 and VM2. With this motivation, we define an operator called interleaving, denoted by $||$. Interleaving is formally defined as follows:

$a||\epsilon=\epsilon||a=a \qquad \forall a\in\Sigma$

$a.s||b.t = a.(s||b.t) \cup b.(a.s||t) \qquad \forall a,b\in\Sigma,\ s,t\in\Sigma^*$

Thus, $ab||ac=\{abac,aabc,aacb,acab\}$. This definition can be extended to interleaving between two sets in a natural way, i.e.

A $||$ B $= \{w/\exists s\in A \wedge t\in B, w\in s||t\}$

For example, consider two sets A and B as follows: A $= \{ab,c\}$ and B $= \{ba\}$ then A $||$ B $= \{abba,abab,baab,baba,cba,bca,bac\}$.

Note that similar to A $||$ B, we also get a set A $||$ A $= \{aabb,abab\}$. We denote A $||$ A by $A^{(2)}$. We use parentheses in the exponent to distinguish it from the traditional use of the exponent i.e. $A^2=A.A$.

Properties of $||$

(1) Interleaving is commutative, i.e.,

$$A \,||\, B = B \,||\, A$$

(2) Interleaving is associative, i.e.,

$$A \parallel (B \parallel C) = (A \parallel B) \parallel C$$

(3) Epsilon is the identity of interleaving, i.e.

$$A \parallel \{\epsilon\} = A$$

(4) The null set is the zero of interleaving, i.e.

$$A \parallel \phi = \phi$$

(5) Interleaving distributes over choice, i.e.

$$(A+B) \parallel C = (A \parallel C)+(B \parallel C)$$

Due to the fifth proposition, we will use A+B || C to mean A+(B || C) rather than (A+B) || C. We also give higher precedence to || than ".". Therefore A.B || C would mean A.(B || C) rather than (A.B) || C. It is easy to see that the . does not distribute over || and vice-versa. The use of the || operator generally results in a set which is exponentially bigger than its arguments. In terms of cardinality we note that

(1) $|A+B| \leq |A| + |B|$ (where + is the arithmetic sum)

(2) $|A.B| \leq |A|.|B|$ (where . is the arithmetic product)

(3) $|A||B| \leq \Sigma \frac{(|x|+|y|)!}{|x|!|y|!}$ $\forall x \epsilon A \wedge y \epsilon B$. This operator, however, does not increase the modeling power of concurrent regular expressions as shown by the following Lemma.

Lemma 4.1: Any expression that uses || can be reduced to a regular expression without || .

Proof: The interleaving of two regular expressions is also a regular expression [Hopcroft 79]. *Q.E.D.*

For example $(ab||bc)$ can be written as $(abbc+abcb+babc+bcab)$ and $a||b^*$ can be written as b^*ab^*.

2.4. Alpha Closure - α

Consider the behavior of people arriving at a supermarket. We assume that the population of people is infinite. If each person CUST is defined as (enter.buy.leave), then the behavior of the entire population is defined as interleaving of any number of people. With this motivation, we define an analogue of a Kleene-Closure for the interleaving operator, alpha-closure of a set A, denoted by A^α as follows:

$$A^\alpha = \bigcup_{i=0,1,..} A^{(i)}$$

In the above example, CUST = $(enter.buy.leave)$ $CUST^\alpha$ = $\{w \mid w \in (enter+buy+leave)^*, \#enter \geq \#buy \geq \#leave$ for any prefix $, \#enter = \#buys = \#leave\}$

We use # a to mean the number of occurrences of symbol a in any string. Thus if a string = {aabba} then #a = 3 and #b = 2.

Note the difference between Kleene closure and alpha closure. The language shown above cannot be accepted by a finite state machine. This can be shown by the use of the pumping lemma for finite state machines [Hopcroft 79]. We conclude that alpha closure can not be expressed using ordinary r.e. operators.

Intuitively, the alpha closure lets us model the behavior of an unbounded number of identical independent sequential agents.

As Σ^* forms a monoid under . (concatenation), Σ^α forms a commutative monoid under the operation $||$. This is because it is closed under $||$, $||$ is commutative and associative, and $\{\epsilon\}$ is left and right identity. As Kleene closure makes a set closed with respect to concatenation, alpha closure makes a set closed under interleaving. We will use this intuition to provide an alternative definition of alpha closure.

Definition: A set **A** is called *closed under interleaving*, or simply *i-closed*, if for any two strings s_1 and s_2 (not necessarily distinct) that belong to **A**, $s_1||s_2$ is a subset of **A**. By definition ϵ must also belong to an i-closed set.

Examples: $\{\epsilon\}$, $\{\epsilon, a, a^2, a^3..\}$, $\{s \mid \#a = \#b\}$ are example of i-closed sets. As Kleene closure of a set A is the smallest set containing A and closed under concatenation, alpha closure of a set A is the smallest set containing A and closed under interleaving. More formally,

Theorem 4.1: Let A be a set of strings. Let B be the smallest *i-closed* set containing A. Then B = A^α.

Proof: A^α contains A and is also i-closed. Since B is smallest set with this property, we get $B \subseteq A^\alpha$.

Since B is i-closed and it contains A, it must also contain $A^{(i)}$ for all i. This implies that B contains A^α. Combining with our earlier argument we get B = A^α. *Q.E.D.*

The above theorem tells us that as Kleene closure captures the notion of doing some action any number of times in series, alpha closure captures the notion of *doing some action any number of times in parallel*. Note that if a set A is i-closed, it is also concatenation closed. This is because if s_1 and s_2 belong to A then so does $s_1||s_2$, and in particular $s_1.s_2$. The following corollary provides us a method of finding A^α by showing that the set is i-closed.

Corollary: A set A is i-closed if and only if A = A^α.

Proof: If A is i-closed, it is also the smallest set containing A and i-closed. By Theorem 4.1, it follows that A = A^α.

Conversely, A = A^α and A^α is i-closed therefore A is also i-closed. *Q.E.D.*

The above corollary tells us that if a set is i-closed, then its alpha closure is the same as itself. As an application of this corollary, we get $A^{\alpha^\alpha}=A^\alpha$.

The following Theorem tells us that interleaving, Kleene closure and alpha closure of i-closed sets remain i-closed. Combining Theorem 4.2 with the previous corollary, we can find alpha closure of sets that are built of some i-closed sets.

Theorem 4.2: If A and B are i-closed then so are A || B,A*,A^α.

Proof:

1) A || B: Let s_1 and s_2 belong to A || B. We will show that $s_1||s_2$ is a subset of A || B.

$s_1 \in p_1||q_1$ because s_1 belongs to A || B , for some $p_1 \in A, q_1 \in B$.

$s_2 \in p_2||q_2$ because s_2 belongs to A || B , for some $p_2 \in A, q_2 \in B$.

$\therefore s_1||s_2 \subseteq p_1||q_1||p_2||q_2$

$= p_1||p_2||q_1||q_2$ (|| is associative and commutative)

$= p||q$ *where* $p=p_1||p_2$ and q = $q_1||q_2$

$\subseteq$ A || B (because p $\subseteq$ A and q $\subseteq$ B as A and B are i-closed)

2) **A***: Let s_1 and s_2 belong to A*. We will show that $s_1||s_2$ is a subset of A*.

$s_1 = p_1.p_2.p_3....p_n$ where each p_i belong to A

$s_2 = q_1.q_2.q_3....q_m$ where each q_i belong to A

Then $s_1||s_2 = p_1..p_n||q_1..q_m \subseteq p_1||..p_n||..q_1||..q_m$

$\subseteq$ A (A is i-closed)

$\subseteq$ A*

3) A^{α}: From Theorem 4.1. *Q.E.D.*

Applying Theorem 4.2, we can easily deduce the following identities.

1) $(A||B)^{\alpha}=A||B$ if A and B are i-closed

2) $A^{*\alpha}=A^*$ if A is i-closed

For example, let $CUST_A$ and $CUST_B$ be sets of strings denoting behavior of customers in supermarket A and B respectively. Both $CUST_A$ and $CUST_B$ are i-closed and therefore, by Theorem 4.2 $CUST_A||CUST_B$ is also i-closed.

The above theorem also tells us that the set of all i-closed sets forms a commutative monoid under the operation ||. This is because they are closed under || , || is commutative and associative, and $\{\epsilon\}$ is left and right identity of this set. As shown below, the other binary operations defined so far do not retain this property.

Theorem 4.3: If A and B are i-closed then A+B and A.B may not be so.

Proof:

1) **A+B**: Consider A = $\{ab\}^{\alpha}$, B=$\{bc\}^{\alpha}$. Let s_1=ab and s_2=bc. Both s_1 and s_2 are members of A+B but s=$abbc \in s_1||s_2$ does not belong to A+B.

2) **A.B**: Consider A $= \{ab\}^{\alpha}$,$B=\{bc\}^{\alpha}$. Let $s_1=abbc$ and $s_2=abbc$ Both s_1 and s_2 are member of A.B but $s=abbcabbc \in s_1 || s_2$ does not belong to A.B.

Q.E.D.

Properties of alpha

1) $A^{\alpha\alpha}=A^{\alpha}$ (idempotence)

2) $(A^{*})^{\alpha}=A^{\alpha}$ (absorption of *)

So far, we have five operations on sets of sequences. These are +,.,*, ||, α. Table 4.2 lists the class of languages generated by using some important subsets of these operators.

Operators	Languages
+,., ||	finite languages
+,.,*, ||	regular languages
+,.,*, || , constrained use of α	unit languages

Table 4.2: Operators and Languages

2.5. Synchronous Composition

To provide synchronization between multiple systems, we define a composition operator denoted by []. Intuitively, this operator ensures that all events that belong to two sets occur simultaneously. For example consider a vending machine VM described by the expression $(coin.choc)^{*}$. If a customer CUST wants a piece of chocolate he must insert a coin. Thus the event *coin* is shared between VM and CUST. The complete system is represented by VM[]CUST which requires that any shared event must belong to both VM and CUST. Formally,

$$A[]B = \{w \mid w/\Sigma_A \in A, w/\Sigma_B \in B\}$$

w/S denotes the restriction of the string w to the symbols in S. For example, $acab/\{a,b\} = aab$ and $acab/\{b,c\} = cb$. If A = $\{ab\}$ and B = $\{ba\}$, then A[]B = ϕ as there cannot be any string that satisfies ordering imposed by both A and B. Consider another set C = $\{ac\}$. Then A[]C = $\{abc, acb\}$.

Properties of []

Many properties of [] are the same as those of the intersection of two sets. Indeed, if both operands have the same alphabet then [] is identical to intersection.

(1) A[]A = A *(Idempotence)*

(2) A[]B = B[]A *(Commutativity)*

(3) A[](B[]C) = (A[]B)[]C *(Associativity)*

(4) A[]NULL = NULL, NULL = (Σ_A, ϕ) *(zero of [])*

(5) A[]MAX = A, MAX = $(\Sigma_A, \Sigma_A{}^*)$ *(identity of [])*

(6) A[](B+C) = (A[]B)+(A[]C) *(Distributivity over +)*

We next show that [] is a well behaved operator in the sense that on combining two i-closed sets with [], the resulting set is also i-closed.

Theorem 4.4: If A and B are i-closed then so is A[]B.

Proof: Let s_1 and s_2 belong to A[]B. Then

$s_1/\Sigma_A \in A$ and $s_1/\Sigma_B \in B$.

Similarly, $s_2/\Sigma_A \in A$ and $s_2/\Sigma_B \in B$.

We will show that $s_1 || s_2/\Sigma_A \subseteq A$ and $s_1 || s_2/\Sigma_B \subseteq B$.

$s_1 || s_2/\Sigma_A$

$=s_1/\Sigma_A || s_2/\Sigma_A$ (Restriction distributes over $||$)

$\subseteq A$ (A is i-closed)

and similarly, $s_1 || s_2/\Sigma_B = s_1/\Sigma_B || s_2/\Sigma_B \subseteq B$

Therefore, $s_1 || s_2 \subseteq A [] B$. *Q.E.D.*

Consider, for example, the set of strings denoting the behavior of customers at a supermarket. That is, POP = $\{enter.buy.leave\}^\alpha$. Now assume that for buying an item a customer has to interact with the sales clerk whose behavior can be written as CLERK = $\{buy\}^*$. Form Theorem 4.4 we conclude that POP [] CLERK is an i-closed set.

3. Modeling of Concurrent Systems

In this section, we give some examples of use of concurrent regular examples in modeling concurrent systems.

Example: $(abc)^\alpha$ [] $a^*b^*c^*$ accepts the language $\{a^n b^n c^n \mid n \geq 0\}$. Note how the use of α operator let us keep track of the number of different symbols that have been seen in the string. This example shows that the strings that can not be recognized even by push down automata can be represented by c.r.e's.

Example: Consider a ball room where both men and women enter, dance and exit. Their entry and exit need not be synchronized but it takes a pair to dance. We would also like to ensure that the number of women in the room is always greater than or equal to the number of men, since idle men can be dangerous! This system can be easily represented using a concurrent regular expression:

A man's actions can be represented by the following sequence:

man :: *menter dance mexit*

A woman's actions as follows:

:

woman :: *wenter dance wexit*

The constraint that the number of women always be greater can be expressed as:

constraint :: $(wenter\ (menter\ mexit)^*\ wexit)^\alpha$

Since any number of men and women can enter and exit independently (except for the constraint) the entire system is modeled as follows:

man^α [] $woman^\alpha$ [] *constraint*

Example: Consider the office of the Department of Motor Vehicles. Two types of clients need service, those who need to get their picture ID taken and those who need to take a test. Clients who need their picture taken first pay the fee and then get their picture taken. Those taking the test, first pay the fee, then take the test and then receive the results of the test. Let us say that there are two clerks - John and Mary - who serve the clients. John receives the fee and Mary hands out results. However the camera is so complicated that it requires both John and Mary to operate it.

The relevant CRE's are :

client1 :: *fee picture*

client2 :: *fee test result*

John :: $(fee + picture)^*$

Mary :: $(result + picture)^*$

DMV :: $((client1)^o \,||\, (client2)^o) \,[]\, (John) \,[]\, (Mary)$

4. Relationship between Concurrent Regular Expressions and STOCS

In this section, we show that any STOCS machine can be converted to its equivalent concurrent regular expression and vice-versa. We need to show the following Lemmas to prove the result establishing the equivalence of STOCS and concurrent regular expressions.

Lemma 4.2: Any unit with multiple *-places can be converted to an equivalent unit with a single *-place (see Figures 4.2(a) and 4.2(b)).

Proof: Let U be a unit with multiple *-places. We construct U', a unit with a single *-place by merging all *-places into a single *-place. All input arcs and output arcs in the previous units are combined. If, in the resulting unit, there is more than one arc with the same label between two places then only one of them is retained. Since the tokens in *-places do not change and the bag of transitions enabled for any configuration is identical for U and U', we conclude that the language accepted by U is the same as the language accepted by U'. *Q.E.D.*

Lemma 4.3: Any unit U is equivalent to another unit U' which has at most two connected components - one with *-place and the other with a single token (see Figures 4.2(b) and 4.2(c)).

Proof: From Lemma 4.2, we can assume, without loss of generality, that there is at most one *-place in U. U may have one or more connected components. Let the connected component C have the *-place. C may have tokens at some simple places too. As tokens move independently of each other within a unit, C can be written as two components- one with tokens only in the simple place and the other with the *-place but no tokens in the simple place. We claim that all the connected components with no *-places can be combined into a single connected component - a finite state machine. This is because there is a finite number of tokens residing in finite places, resulting in only a finite number of possible configurations. There is an edge labeled a from configuration C_1 to C_2 if and only if configuration C_1 can result in C_2 after making a transition a. A finite state machine can be simulated by a connected component with a single token in its initial state. *Q.E.D.*

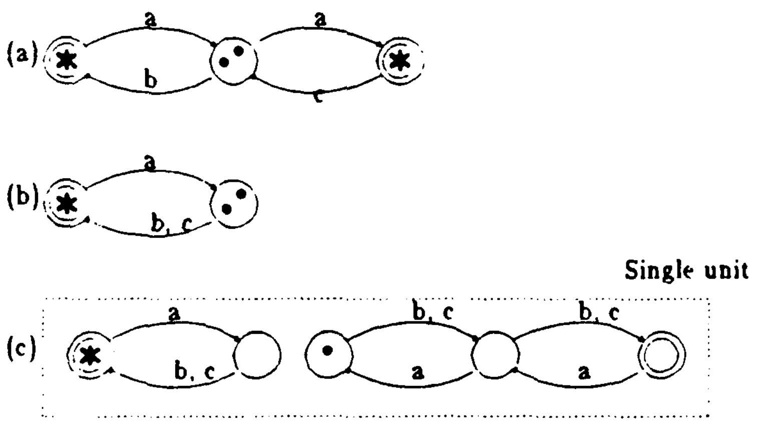


Figure 4.2 : Lemma 4.2 and 4.3

Lemma 4.4: Let U be a unit with a single *-place having no tokens in its simple places. Then its language can be written as a (*regular expression*)$^{\alpha}$.

Proof: Let $U=(P,C,\Sigma,\delta,F)$ with $C(p_i) = *$. We construct the finite state machine $A=(P,p_i,\Sigma,\delta,F)$. Let L(X) represent the language accepted by automata X. We will show that $L(U)=L(A)^{\alpha}$.

Case 1: $L(U)\subseteq L(A)^{\alpha}$

Let a string s belong to the language of the unit U. In accepting s, a finite number of tokens, say n, must have moved from the *-place to some final place. Let $s_1.s_2..s_n$ be the strings that are traced by tokens 1..n, respectively, such that one of their interleaving is s. Each of the strings $s_1..s_n$ also belongs to the regular set. Therefore, their interleaving belongs to alpha-closure of the regular set.

Case 2: $L(A)^\alpha \subseteq L(U)$

Consider any string s in $L(A)^\alpha$. This string s can be written as $s_1||s_2||..||s_n$ where each s_i belong to A. As s_i belong to A, it also represents a path from the initial place to a final place in U. Hence s can be simulated by n tokens which simulate $s_1,..s_n$ respectively. *Q.E.D.*

Theorem 4.5: There exists an algorithm to derive a concurrent regular expression that describes the set of strings accepted by a STOCS machine.

Proof:

Clearly it is sufficient for us to derive a concurrent expression for each unit, as the concurrent expression equivalent to the STOCS will be the concurrent regular expressions for units composed by the [] operator.

To derive the expression for a unit, we use Lemma 4.3 to convert it into a unit with at most two components, one with *-place and one with a single token. From Lemma 4.4, the language of any such unit can be written as interleaving of a regular expression and at most one (*regular expression*)$^\alpha$.

For example, to describe the language of the unit shown in Figure 4.2(a), we first convert it to 4.2(b) by Lemma 4.2. We convert the unit in 4.2(b) to 4.2(c) by Lemma 4.3. There are two connected components in the unit of Figure 4.2(c). The regular expression for the first component with * replaced by a single token is (a.(b+c))*. The regular expression for the second component is $(((ba+ca)^*.a)^*.((ba+ca)^*.(b+c))$. Thus, the expression for the entire unit can be written as $(a.(b+c)^{\alpha}||(((ba+ca)^*.a)^*.((ba+ca)^*.(b+c))$ Before we prove the converse of the above Theorem, we need the following Lemmas.

Lemma 4.5: $(A||B)^{\alpha} =$

$(A||B)$ if both A and B are i-closed

$(A||B^{\alpha})$ if A is i-closed

$(A^{\alpha}||B)$ if B is i-closed

C^{α} where C is a regular set if both A and B are regular sets

Proof:

(1) *Both A and B are i-closed.*

By Theorem 4.2, A || B is i-closed.

$=>$ $(A||B)^{\alpha}$ = A || B by Theorem 4.1.

(2) *Only A is i-closed.*

$(A||B)^{\alpha}$=$(A^{\alpha}||B)^{\alpha}$ (because A^{α}=A)

We will show that $(A^{\alpha}||B)^{\alpha}$=$A||B^{\alpha}$

We first show that $s \in (A^{\alpha}||B)^{\alpha} \Rightarrow s \in A||B^{\alpha}$

let $s \in (A^{\alpha}||B)^{\alpha}$

$\Rightarrow s \in s_1||s_2||s_3..s_m$ where $m \geq 0$.

$\subseteq (a_{11}||a_{12}..||a_{1n_1}||b_1)||(a_{21}||a_{22}||..a_{2n_2}||b_2)..||..(a_{m1}||a_{m2}||..a_{mn_m}||b_m)$

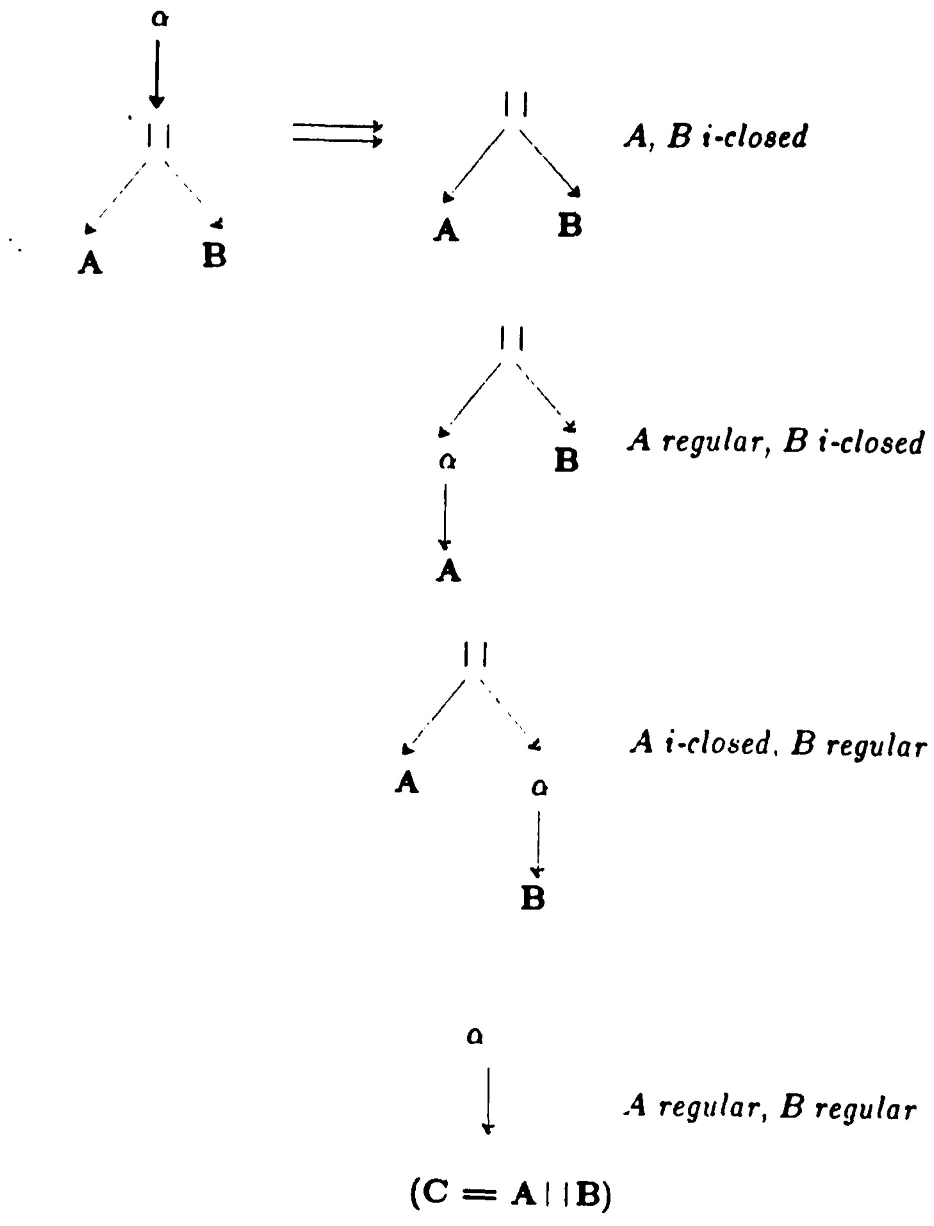


Figure 4.3: Lemma 4.5

where $a_{ij} \in A$ for $i = 1..m$ and $j=1..n_i$

On rearranging terms, s also belongs to $A||B^{\alpha}$

We now show that $s \in A||B^{\alpha} \Rightarrow s \in (A^{\alpha}||B)^{\alpha}$

Let $s \in A||B^{\alpha}$

$=> s \in a||b_1||b_2||b_3..||b_m$ $where m \geq 0$

$\subseteq (a||b_1)||(\epsilon||b_2)....(\epsilon||b_m)$

$\subseteq (A||B)^{\alpha}$

(3) *Only B is i-closed*

Similar to (2)

(4) *Both A and B are regular* $=>$ A || B = C as interleaving of two regular sets is also a regular set. $=> (A||B)^{\alpha}=C^{\alpha}$

Lemma 4.6: Let A and B be two regular expressions, then $A^{\alpha}||B^{\alpha}=(A+B)^{\alpha}$

Proof: Let string $s \in A^{\alpha}||B^{\alpha}$.

$\Rightarrow s \in a_1||a_2||..||a_n||b_1||b_2||..||b_m$ for $a_i \in A, i=1..n$,

$b_j \in B, j=1..m \quad n,m \geq 0$

$\subseteq (A+B)^{\alpha}$ (because each string belong to A+B)

Let string $s \in (A+B)^{\alpha}$.

$\Rightarrow s \in c_1||c_2||..||c_n$, where $c_i \in A+B$

If $c_i \in A$ we call it a_i, otherwise we call it b_i

on rearranging terms so that all strings that belong to A come before strings that do not belong to A (and therefore must belong to B), we get $s \in A^{\alpha}||B^{\alpha}$*Q.E.D.*.

Lemma 4.7: Any unit expression U is equivalent to another unit expression which is the interleaving of regular expressions and $(\textit{regular expression})^{\alpha}$. Expressions in these forms are called *normalized unit expressions*.

Proof: To show this Lemma, we will use induction on the number of times || or α occurs in a unit expression. The Lemma is clearly true when the expression does not have any occurence of || or α as a regular expression is always normalized. Let U be a expression with at most k occurrences of || or α.

Then U can be written as $U_1||U_2$ or U_1^α where U_1 and U_2 can be normalized by the induction hypothesis. We will show that U can also be normalized.

(1) $U=U_1 \,||\, U_2$

$U_1=A_1||B_1{}^\alpha$ where A and B are some regular expressions

$U_2=A_2||B_2{}^\alpha$ where A and B are some regular expressions

Therefore, $U_1||U_2=(A_1||B_1{}^\alpha)||(A_2||B_2{}^\alpha)$

$=(A_1||A_2)||(B_1{}^\alpha||B_2{}^\alpha)$ (|| is associative and commutative)

$=(A_1||A_2)||(B_1+B_2)^\alpha$ (by Lemma 4.6)

$\therefore$ U can be normalized.

(2) $U=U_1^\alpha$

$U=U_1^\alpha=(A||B^\alpha)^\alpha$

where A and B are some regular expressions.

Since B^α is i-closed and A is a regular set from Lemma 4.5, we obtain,

$U=A^\alpha||B^\alpha$

$=(A+B)^\alpha$ (from Lemma 4.6)

$=C^{\prime\alpha}$ for some regular expression C.

$\therefore$ U can be normalized. *Q.E.D.*

Theorem 4.6: There exists an algorithm to derive a STOCS machine that describes the set of strings described by a concurrent regular expression.

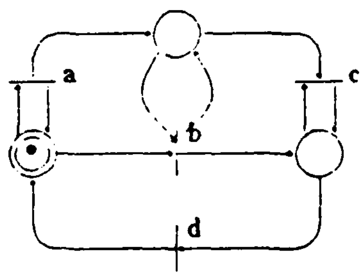
Proof: Any regular expression can be converted to a finite state machine by standard techniques as described in [Hopcroft 79].

Using Lemma 4.7 we can convert any unit expression into the normalized form. To convert a normalized unit expression into a unit we simply use a finite state machine for each regular expression and use Lemma 4.5 to construct a connected component with a *-place for (*regular expression*)$^{\alpha}$. STOCS is just the union of all units constructed by the above procedure. Clearly, the STOCS so constructed accepts the same language as characterized by the given concurrent regular expression. *Q.E.D.*

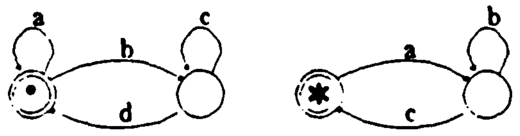
Thus, the class of languages accepted by STOCS and concurrent regular expressions is identical.

Theorem 4.6 provides us the flexibility of specifying a system in terms of concurrent regular expressions and then converting it to a Petri Net which can be analyzed for function correctness using the coverability tree[Karp 68], reachability algorithm[Mayr 86] and matrix equations[Murata 84]. Figure 4.4(a) shows an example of a Petri net which is converted to a STOCS machine shown in Figure 4.4(b). The concurrent regular expression equivalent to the Petri net is obtained using the STOCS machine and is shown in Figure 4.4(c).

(a)



(b)



(c) $(a+bc^*d)^* \; [] \; (ab^*c)^{\alpha}$

Figure 4.4 : FLOPN => STOCS => CRE

5. Concurrent Regular Languages

From our earlier results we know that the class of language accepted by STOCS is identical to that characterized by concurrent regular expressions. The definition of concurrent regular expression is hierarchical as a concurrent regular expression is defined using unit expressions which are defined using regular expressions. Regular languages are those set of strings that can be accepted by a regular expression. Unit languages are those set of strings that can be accepted by a single unit expression. In this section, we show that the regular languages are properly contained in the unit languages which are properly contained in the concurrent regular languages.

5.1. Regular Languages

Theorem 4.7: The unit languages properly contains the regular languages.

Proof: As a finite state machine is also a unit with a single token, unit languages contain regular languages. To see that the inclusion is proper, consider the language $\{(a.b)^{\alpha}\}$, which is accepeted by a unit in Figure 4.5, but is not accepted by a finite state machine.

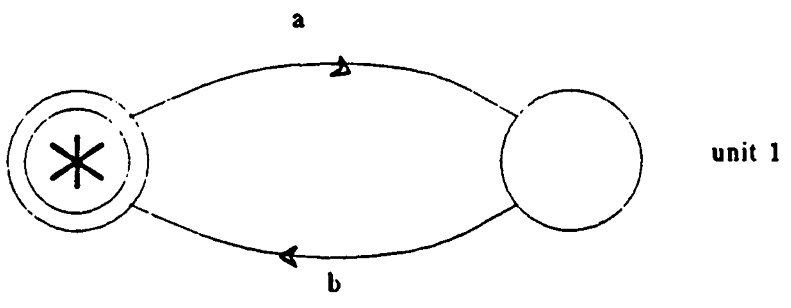


Figure 4.5: A unit machine for $(ab)^{\alpha}$

5.2. Unit Languages

All unit languages are also concurrent regular languages. We next show that this containment is also proper.

Definition: A language is called *i−open* if there does not exist any non-null string s such that if t belongs to a language then so does $s||t$.

Example: All finite languages are i-open. a^*,$(a+b)^*$,$(ab)^\alpha$ are not i-open because a,aba,andab are strings respectively such that their interleaving with any string in the languge keeps it in the languge. Recall that i-closed languages are set of strings that are closed under interleaving. All i-closed languages are not i-open and all i-open languages are not i-closed. However, there are languages that are neither i-open nor i-closed. An example is $a^*b^*||c^*$ which is not i-open as any interleaving with c keeps a string in the language. It is not i-closed because $abc||abc$ does not belong to the language.

Theorem 4.8: A unit cannot accept a non-regular i-open language.

Proof: Let L be a non-regular i-open language. Since this language is not accepted by a finite state machine, the unit should have a *-state. For the similar reason tokens, must move out of the *-state and must eventually reach a final state. This implies that there exists a non-null path p from the *-state to a final state. This implies that for any string t that belongs to L, $t||p$ will also belong to L, a contradiction because L is an i-open language. *Q.E.D.*

For example, consider the language a^nb^n. A 2-stocs for this language is shown in Figure 4.6. The language is i-open because there is no non-null string, such that its indefinite interleaving exists in the language. By Theorem

4.8, we cannot construct a single unit to accept this language.

Theorem 4.8 tells us that unit languages are properly contained in STOCS languages. Our example shows that there exists a STOCS machine with two units - one with a *-state and the other without - which can not be accepted by a single unit. Now we show that there exists two units both with *-state which cannot be recognized by a single unit.

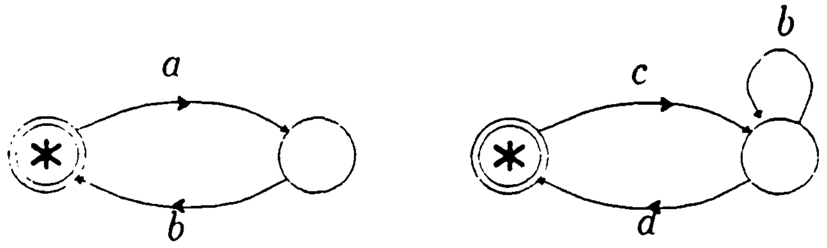


Figure 4.6: A STOCS machine for $(a_1b_1)^\alpha[](a_2b_1{}^*b_2)^\alpha$

Theorem 4.9: There are i-closed concurrent regular languages that cannot be accepted by a unit.

Proof: Consider the language L $= (a_1b_1)^\alpha[](a_2b_1{}^*b_2)^\alpha$ which can obviously be recognized by 2-STOCS. Assume if possible that it can be recognized by a single unit. Since $\epsilon \in L$, there are no tokens in the non-final places. Therefore any string that traces a path from a *-place to a final place is also a member of L. We show that there exists a string which is not a member of L and which traces a path from a *-place to a final place.

$a_2a_1{}^nb_1{}^nb_2 \in L$ but $a_2a_1{}^{n+k}b_1{}^nb_2$ does not belong to L for any $k>0$. This implies that while making transitions on a_1 the symbols must move out of *-place. This implies that there is a path from the *-place to the final place which starts with a_1. Therefore, the machine also accepts a string starting with the symbol a_1. No such string belongs to the language. *Q.E.D.*

From above discussion, we note that

regular $\subset$ unit $\subset$ STOCS

6. Conclusions

In this chapter, we have defined an extension of regular expressions called concurrent regular expressions and have shown that they can be transformed to STOCS machines. The equivalence of STOCS machines and CRE formalism is comforting as we can specify in one formalism and analyze in the other. The concurrent regular expression is built of regular expressions and operators - interleaving, alpha closure and synchronous composition. These operators concisely capture two notions of distributed systems: concurrency and synchronization.

CHAPTER 5

Comparison With Petri Nets

1. Introduction

In this chapter, we do a detailed comparison of STOCS machines with Petri nets. There are two reasons for choosing Petri nets for comparison. First, Petri nets have been used extensively in the design and analysis of concurrent programs and are considerably more popular than, say, UCLA graphs or computation graphs. Second, we have shown in this chapter that, loosely speaking, the power of the STOCS model is the same as that of Petri nets. Thus, it would be unfair to compare the STOCS model with, say, the Finite State Machine Model, which is less powerful, or PRAM, which is more powerful than the STOCS model.

This chapter is organized as follows. Section 2 compares the complexity of reachability in the Petri net and the STOCS model. Section 3 compares them for ease in modeling of concurrent systems. Section 4 compares their languages.

2. Comparison of Reachability

In this section, we show that the reachability problem is equivalent for Petri nets and STOCS machines. This gives us confidence that systems that are modeled as configurations of a Petri net can equivalently be modeled as configurations of STOCS machines. Instead of showing the equivalence of STOCS machines with general Petri nets, we show their equivalence with ordinary Petri nets. An ordinary Petri net is a special case of a general Petri net with the restriction that no place has multiple input (or output) arcs to the same transition. We can restrict our focus to ordinary Petri nets because of the following Lemma due to Hack.

Lemma 5.1 [Hack 76]: The reachability problem is equivalent for general Petri nets and ordinary Petri nets.

Proof: Hack provides a construction to convert a general Petri net to an ordinary Petri net such that the reachability problem is equivalent. This construction replaces a place with maximum multiplicity of k by a ring of k places each having multiplicity of 1 (see Figure 5.1). *Q.E.D.*

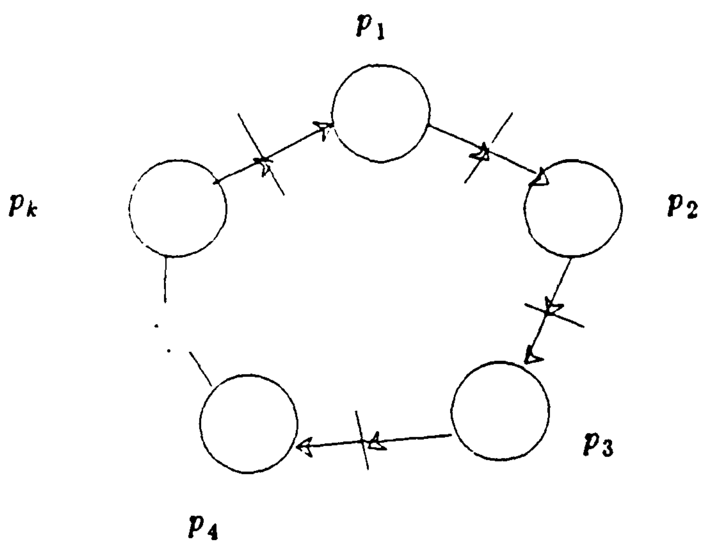


Figure 5.1: General Petri net => Ordinary Petri net

To show that the reachability problem of an ordinary Petri net is reducible to the reachability problem in a STOCS machine, we will construct an equivalent STOCS structure from a given Petri Net structure. The structures are equivalent in the sense that any configuration that is reachable in one structure is also reachable in the other. We also require that the structure of the STOCS model is no bigger than a constant multiple of the size of Petri net. A single Petri net has multiple STOCS representations, each corresponding to different *unit assignments.* A unit assignment is a mapping from the set of places in Petri nets to the set of natural numbers representing the *unit* numbers. Intuitively, each place in a Petri net is assigned to a process. A unit assignment is called *consistent* if no two places which are input (output) to the same transition have the same unit number. This constraint is required because for every transition in a STOCS machine, there is at most one place per unit that loses (gains) a token. A trivial consistent unit assignment is the one which assigns every place a different number; hence there always exists at least one consistent unit assignment.

Theorem 5.1: Reachability problem of Ordinary Petri nets is reducible in linear time to that of a STOCS Machine.

Proof: *(1) Construction of a STOCS machine from an Ordinary Petri net*

An ordinary Petri net is converted to a STOCS machine as follows. Every place in the Petri net is also a place in the STOCS machine (see Figure 5.2). These places, however, may belong to different units. Let N be a Petri net = (P,T,I,O,M) with the usual meaning of the notation.

We first find a unit assignment function $f:P \rightarrow 1,2,..K$ such that

$$\forall t \in T, p_1,p_2 \in P: \quad (p_1,p_2) \subseteq I(t) \vee (p_1,p_2) \subseteq O(t) \Rightarrow f(p_1) \neq f(p_2).$$

This condition implies that places belonging to the same unit cannot be input(output) to the same transition. It holds trivially if all places belong to different units

We define the STOCS machine S as the set of units U_i where $i=1...K$

Each unit U_i is defined as follows:

$U_i=(P_i,\Sigma_i,C_i,\delta_i)$ † where:

- P_i contains all the places that are assigned the unit number i, and a *-place denoted by sp_i.

  $P_i=\{p\in P \mid f(p)=i\} \cup \{sp_i\}$

- Σ_i contains as handshake symbols all those transitions in which places belonging to unit i participate. It is assumed that each transition has a unique label.

  $\Sigma_i=\{ t\in T \mid \exists p\in P_i,\ p\in I(t)\cup O(t)\}$

- The configuration of the STOCS machine ($C_i:P_i\rightarrow\mathbb{N}$) is the same as the marking function in the Petri net, i.e.

  $C_i(p)=M(p)$ for $p\in P_i$,

  $C_i(sp_i)=*$.

- $\delta_i\subseteq P_i\times\Sigma_i\times P_i$. If a unit has an input place as well as an output place for a transition, an arc is added between them. If a unit has only an input place for a transition then an arc is added between the input place and its *-place. If a unit has only an output place for a transition then an arc is added between its *-place and the output place. Formally,

  $\delta_i=\{(p_j,t,p_k) \mid \exists t,\ p_j\in I(t)\wedge p_k\in O(t)\}$

  $\cup\ \{(p_j,t,sp_i) \mid \exists t\ p_j\in I(t), \nexists\ p_k,\ p_k\in P_i,p_k\in O(t)\}$

  $\cup\{(sp_i,t,p_k) \mid \exists t\ p_k\in O(t), \nexists\ p_j\in P_i,p_j\in I(t)\}$

† We ignore the set of final places as they are irrelevant for the reachability problem.

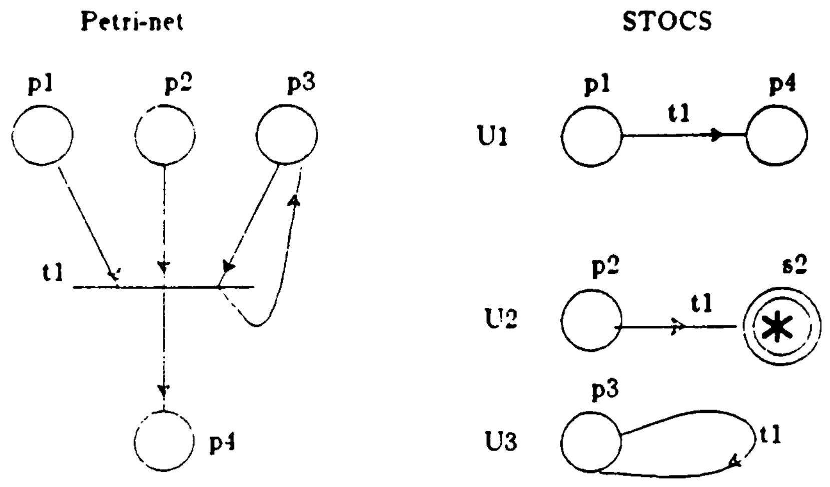


Figure 5.2: Petri net => STOCS Machine Conversion

The size of the resulting STOCS machine is of the same order as the size of the Petri net. Also, the transformation of the given Petri net structure can be done in linear time.

Reachability is equivalent in both structures

We next show that any transition that is enabled in Petri net is also enabled in the STOCS machine and vice-versa. The set of sequences of transitions is identical for both structures because:

(1) *Initially, both the Petri net and the STOCS machine have the same configuration*. Note that while considering the configuration of a STOCS machine we need only consider tokens in simple places, as the tokens in *-places do not change. More formally, $C_i = M \quad \forall i$.

(2) *The set of transitions that is enabled for equal configurations is identical.*

Let t be enabled in Petri Net N.

$=> \forall p \in I(t)$: $M(p) \geq 1$ (by definition of enablement).

We will show that t is also enabled in the STOCS machine.

Let $t \in \Sigma_i$

$=> \exists p_j \in P_i$: $p_j \in I(t) \cup O(t)$. (by definition of Σ_i)

Case 1: $p_j \in I(t)$

$=> C_i(p_j) \geq 1$ (by definition of C_i)

$=> t$ is enabled in C_i.

Case 2: $p_j \in O(t) \wedge \nexists p_k \in P_i : p_k \in I(t)$

$=> (sp_i, t, p_j) \in \delta_i$ (by the definition of δ_i)

$=> t$ is enabled in C_i since $C_i(sp_i) = '*$,

If a transition is enabled in the STOCS machine then it is also enabled in Petri net by a similar argument.

(3) *Both machines starting from equal configurations reach equal configurations on taking the same transition.* On executing the transition t in Petri net, the new marking M' is defined as follows:

$p \in I(t) \Rightarrow M'(p) = M(p) - 1$

$p \in O(t) \Rightarrow M'(p) = M(p) + 1$

otherwise M'(p) = M(p).

The configuration in the STOCS machine can change only in units that have t in their Σ. By the definition of execution in the STOCS machine if $(p_i, t, p_j) \in \delta_i$ then

$C'(p_i) = C(P_i) - 1 = M'(p_i)$

and $C'(p_j) = C(p_j) + 1 = M'(p_j)$.

Q.E.D.

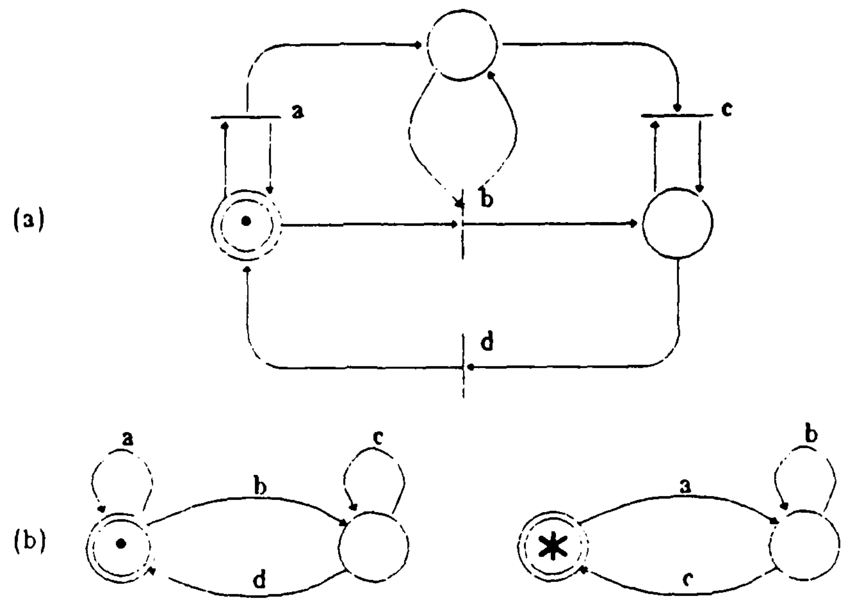


Figure 5.3: Conversion from a Petri net to a STOCS Machine

We now present an example that shows the conversion of ordinary Petri nets to STOCS machines. The Petri net in Figure 5.3 is converted as follows. We assign p_1 and p_2 to the same unit U_1, as they do not share any transition for input or output. p_3 is assigned to U_2. Corresponding to transition a we draw an arc from p_1 to itself in U_1. Since there is no input place for transition a in U_2 but an output place p_3, we draw an arc from the *-place, sp_2 to p_3. A pseudo Pascal procedure to convert a Petri net to a STOCS machine is given in Figure 5.4.

```
Procedure Petri_to_STOCS;
begin

(* Convert general petri net to ordinary petri-nets *)
New_Petri := Hack(Petri); (* using Hack's construction *)

(* find a consistent unit assignment *)
(* returns an array color[] such that no two places
that share a transition are assigned the same color
Let there be d colors *)
Consistent_Unit_Assignment;

(* Construct a STOCS with d units *)
STOCS := (U1,U2,....Ud);
where $U_i=(P_i,\Sigma_i,M_i,\delta_i)$

 (* construct $P_i$'s *)
for all $p_i$ do
 if $color(p_i)=c$ then $P_c:=P_c \cup p_i$ ;

(* for each unit i construct a *-place s(i) for that unit *)
$P_i:=P_i \cup sp_i$;

(* Construct $\Sigma_i$'s *)
for each $t_i$ do
 for each $p_i \in I(t_i) \cup O(t_i)$ do
   c := color($p_i$)
   $\Sigma_c:=\Sigma_c \cup t_i$;

(* Construct $\delta_i$'s *)
for all $t_k$ do
  for all $p_i \in I(t_k)$ do
   c := color($p_i$);
   if $\exists p_j \in O(t_k)$ such that color($p_j$) = c then
            $\delta_c:=\delta_c \cup (p_i,t_k,p_j)$
      else $\delta_c:=\delta_c \cup (p_i,t_k,sp_c)$);
   for all $p_i \in O(t_k)$ do
    if $\nexists$ $p_j \in I(t_k)$ such that
     c:= color($p_i$);
     $\delta_c:=\delta_c \cup (sp_c,t_k,p_i)$;

(* Duplicate Marking *)
C($p_i$) := M($p_i$);

end;
```

Figure 5.4: A Program to convert a PN to a STOCS Machine

Corollary 5.1 : The complexity of the reachability problem for a STOCS machine is at least exponential space.

Proof: This corollary follows from Theorem 3.1 and an earlier result by Lipton [Lipton 76] which shows that the reachability problem for Petri Nets is of at least exponential space complexity.

Conversion of reachability in a STOCS machine to that in a Petri net is complicated because a handshake may occur in a unit multiple times. Thus the conversion provided in Theorem 3.1 cannot be reversed to provide a constructive proof of this Lemma. While converting a STOCS machine to a Petri net, a single handshake is converted to transitions, reflecting all possible ways the handshake could execute. The proof of the Theorem 5.2 shows the procedure formally.

Theorem 5.2: The reachability problem of a STOCS machine is reducible to that of a Petri net.

Proof:

Let S = $(U_1, U_2 ... U_n)$. The Petri Net N = (P, T, I, O, M) where

- $P = \bigcup_{i=1}^{i=n} (P_i - sp_i)$

  The places in the Petri net is the union of all the simple places in STOCS.

- For each handshake symbol in the STOCS machine, we have one or more transitions in Petri net. Let the handshake a occur in a unit i, n_i times. Then the number of transitions required is the product of all n_i's. i.e.

  T = $\{a_S\}$ where

  $S \subseteq \bigcup_{i=1}^{i=n} \delta_i$

$|S \cap \delta_i| = 1$ $\forall i$ $a \in \Sigma_i$ The above condition states that if a handshake belongs to a unit then there is exactly one arc from that unit. That is, we construct a transition for each combination of arcs labeled with that handshake.

- $I(a_S) = \{p_i \in P \mid \exists p_k: (p_i, a, p_k) \in S\}$
- $O(a_S) = \{p_i \in P \mid \exists p_k: (p_k, a, p_i) \in S\}$
- $M(p) = C_i(p)$ if $p \in P_i$.

The set of sequences of transitions is identical for both structures because:

(1) *Initially, both the STOCS machine and the Petri net have the same configuration.* Note that while considering the configuration of a STOCS we need to consider tokens only in simple places as the tokens in *-places do not change. Due to the definition of M, the STOCS machine and the Petri net initially have the same configuration.

(2) *The set of transitions that is enabled in the STOCS machine and the Petri net for equal configurations is identical.*

Let a_S be enabled in Petri Net N.

$\Rightarrow \forall p \in I(a_S)$: $M(p) \geq 1$.

$\Rightarrow$ $C_i(p) \geq 1$ (by definition of $I(a_S)$)

$\Rightarrow$ a_S is enabled in STOCS.

It is also easily verified that a transition enabled in STOCS is also enabled in Petri net.

(3) *Both machines started from equal configurations reach equal configurations on taking the same transition.*

On executing the transition t in Petri net, the new marking M' is defined as follows:

$p \in I(t) \Rightarrow M'(p) = M(p) - 1$

$p \in O(t) \Rightarrow M'(p) = M(p) + 1$

otherwise M'(p) = M(p).

The configuration in STOCS can change only in units that have t in their Σ. By definition of execution in STOCS if $(p_i, t, p_j) \in \delta_i$ then

$C'(p_i) = C(P_i) - 1 = M'(p_i)$

and $C'(p_j) = C(p_j) + 1 = M'(p_j)$. *Q.E.D.*

Figure 5.5 shows such a construction. Corresponding to the handshake *mem* in the STOCS machine, we get the transitions $mem_{(p_1,mem,p_2),(p_5,mem,p_6)}$ and $mem_{(p_2,mem,p_3),(p_5,mem,p_6)}$ The first *mem* corresponds to the handshake between U_1 and U_2 with the tokens at p_1 and p_5 whereas the second *mem* corresponds to the handshake with tokens at p_2 and p_5. *-places are removed. Figure 5.6 gives a procedure written in pseudo Pascal to convert a STOCS machine to a Petri net.

Corollary 5.2: The class of languages accepted by free labeled ordinary Petri nets is identical to that accepted by free labeled STOCS(FLSTOCS) machines.

Proof. In the proof of Theorem 5.1, we assigned a free labeling to the Petri net (the reachability problem in Petri net is independent of its labeling). The resulting STOCS machine accepted the same language as that accepted by the Petri net. In the proof of Theorem 5.2, the number of instances of each handshake is exactly one if the STOCS machine is free labeled. The Petri net

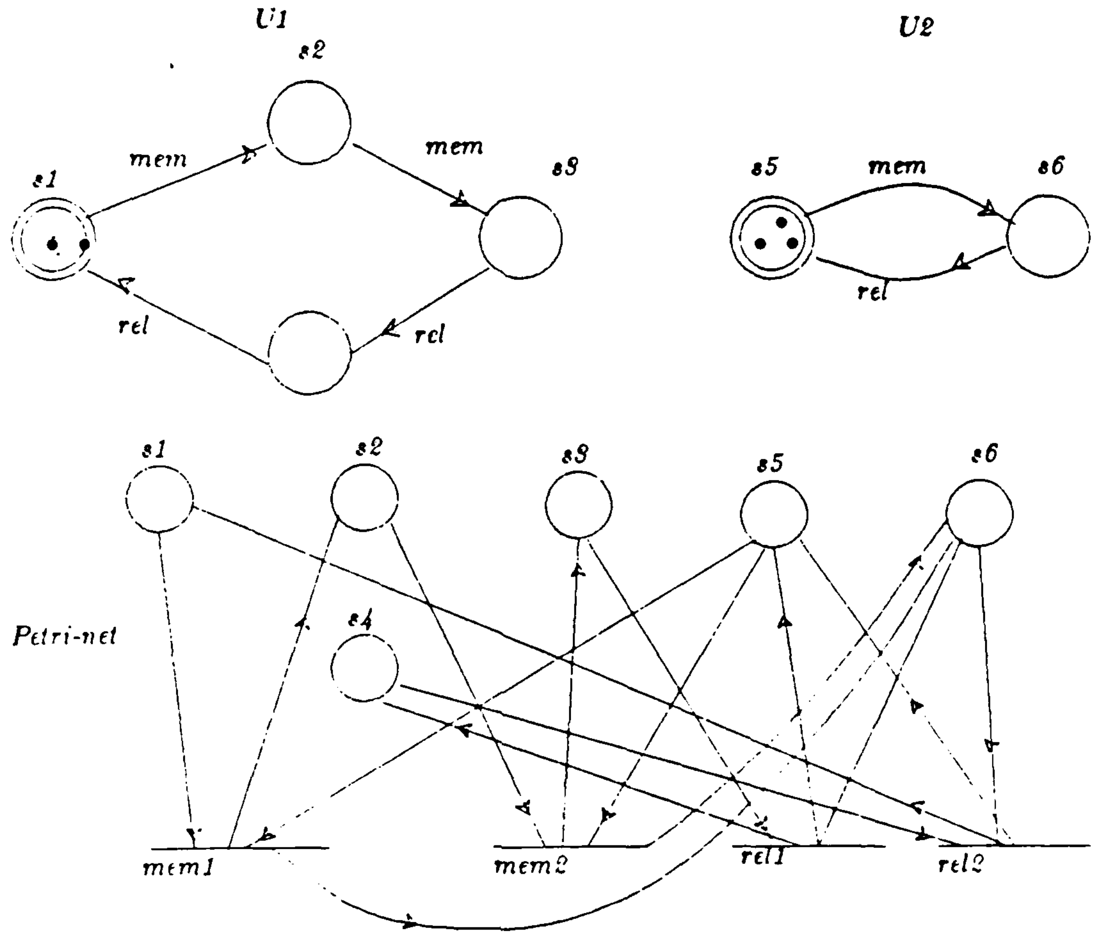


Figure 5.5: Petri Net for 2-out-of-3 problem

constructed out of the STOCS machine has the same language. From these two Theorems, it can be concluded that the class of languages accepted by FLOPN and FLSTOCS machines is identical. *Q.E.D.*

```
Procedure STOCS_to_Petri();
begin

  (* Construct P *)
  P := $\bigcup_i P_i - sp_i$

  (* Construct T *)
  T := $\{a_S \mid where C \subseteq \bigcup \delta_i,$ such that $\mid S \cap \delta_i \mid = 1 \ \forall i\}$

  (* Construct I(a_S) *)
  $I(a_S) = \{p_i \in P \mid \exists p_k: (p_i, a, p_k) \in S\}$

  (* Construct O(a_S) *)
  $O(a_S) = \{p_i \in P \mid \exists p_k: (p_k, a, p_i) \in S\}$

  (* Marking *)
  $M(p) = C_i(p)$ if $p \in P_i$.

end
```

Figure 5.6: A Procedure to Convert a STOCS machine to a PN

2.1. Decomposition of a Petri net: Consistent Unit Assignment

As mentioned earlier, a Petri net has multiple equivalent STOCS machines depending on different unit assignments. In our proof of Theorem 5.1, we used the trivial consistent unit assignment - assignment of each place to a different process. This assignment may result in a large number of units and the resulting STOCS may not be easy to understand. For example, Figure 5.3 shows a Petri net and its equivalent STOCS machine. The alternative machine shown in Figure 5.7 is more difficult to understand and has more *-places than the machine in Figure 5.3.

Since each unit represents a completely independent entity, a reasonable measure of complexity of a STOCS machine is the number of units it contains. With this motivation, it is useful to convert a Petri net into a STOCS machine

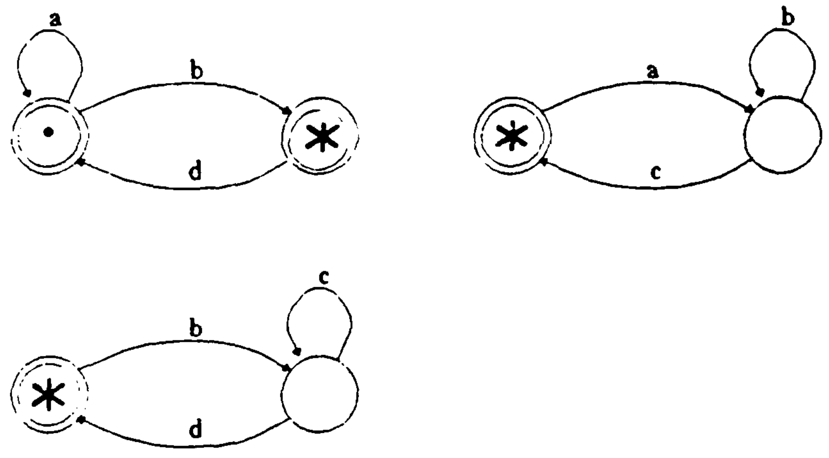


Figure 5.7: An alternate STOCS machine for Petri net in Figure 5.3

such that the number of units is minimized. We first show that the problem of finding a consistent unit assignment such that the number of units in the resulting STOCS machine is minimum is NP-complete [Garey 79]. More formally,

Theorem 5.3: The following problem is NP-complete.

Instance: An ordinary unmarked Petri net N = (P,T,I,O) and a positive number K <= |P|.

Question: Is Petri net N, K-decomposable, i.e. is there a function $f: P \rightarrow \{1,2 \ldots K\}$ such that

$\forall t: (p_1,p_2) \subseteq I(t) \vee (p_1,p_2) \subseteq O(t) \Rightarrow f(p_1) \neq f(p_2)$.

Proof:

(a) It is in NP.

This is immediate as there is a succinct certificate of K-decomposability - the consistent unit assignment function. In other words, a Turing machine can non-deterministically guess the unit assignment function and proceed to verify that it is consistent

(b) Reduction from vertex coloring to K-decomposability.

Let there be a graph G=(V,E). We construct a Petri net N from it as follows:

N = (P,T,I,O) where

P = V

T = E

$I(t) = \{(v1,v2) \mid t=(v_1,v_2)\}$

$O(t) = \phi$

Assume that this Petri net is K-decomposable. This implies that there exists a function f:V->{1,2,..K} such that

$f(v_1)=f(v_2) \Rightarrow \nexists t(v_1,v_2) \in I(t)$.

This condition is identical for K-coloring of the original graph. Therefore, it follows that N is K-decomposable iff G is k-colorable. *Q.E.D.*

The above proof shows how K-colorability can be transformed into K-decomposability. Figure 5.8(a) illustrates this.

We next show that K-decomposability of a Petri net can be reduced in linear time to K-colorability of a graph. Therefore, we can use any algorithm that returns good sub-optimal coloring or optimal coloring with good probability to solve K-decomposability problem.

Theorem 5.4: K-decomposability of a Petri net can be reduced to K-colorability of a graph.

Proof: We construct a graph G = (V,E) as follows.

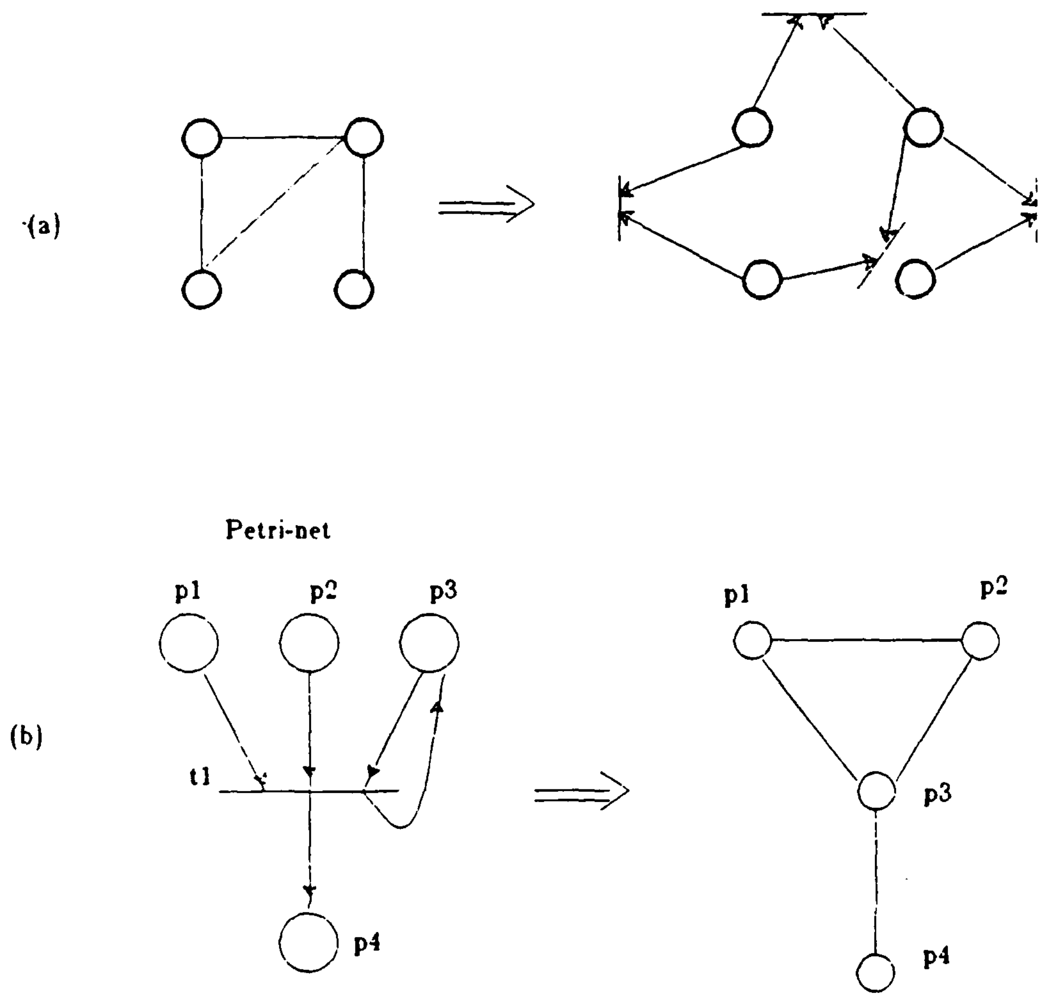


Figure 5.8: K-decomposability <=> K-colorability

$V = P$, the set of places

$E = \{(v_1,v_2) \mid \exists t \ \{v_1,v_2\} \subseteq I(t) \vee \{v_1,v_2\} \subseteq O(t)\}$

Clearly, if there exists a K-color assignment to the graph, the original Petri net is K-decomposable.

Figure 5.8(b) shows a conversion from K-decomposability of a Petri net to K-colorability of a graph. Note the conversion of an ordinary Petri net such that each transition has exactly one input and one ouput. Such a Petri net is equivalent to a finite state machine. When such a Petri net is converted to a

STOCS machine, a single unit is enough as consistency conditions are always satisfied. The reduction is pleasant as it gives us back the classical finite state machine. Another observation is that any unit without *-places is just an S-invariant of the original Petri net [Murata 84]. Thus, a Petri net can always be decomposed into S-invariants and units with *-places.

3. Comparison of Ease in Modeling

Having compared the inherent power of both models, we now compare the convenience of modeling in them. Petri nets have been used to model a large variety of systems such as computer hardware, computer software, PERT, chemical equations and communication protocols. Our aim is to analyze concurrent systems and we will limit our discussion accordingly. We further constrain our modeling to systems that use synchronous messages. We believe that concurrent systems with synchronous messages are easier to model using STOCS machines than Petri nets for the following reasons:

1) The STOCS model is closer to programming languages.

Once a concurrent system has been specified in a concurrent model, it needs to be implemented in some programming language by filling the details missing in the high-level specification. It is easier to derive an implementation from a model that is closer to a programming language. The STOCS model is closer to most concurrent programming languages than a Petri net is. Most concurrent programming languages use synchronous communication which is closer to the semantics of a handshake in STOCS. Specifically, rendezvous of Ada and I/O statement of CSP can be easily represented in the STOCS model. The STOCS machine also has the notion of process(unit) which is missing in Petri nets. As a result of this closeness to programming languages, we have incorporated the STOCS formalism in C to make it suitable for concurrent programming. This aspect of the STOCS model is discussed further in Chapter 8.

2) Petri nets require an explicit specification of interaction between multiple processes.

The disadvantage of Petri net's style of modeling is that a net can become very complicated because places belonging to different processes get intermixed. The implicit interaction between processes based on the name of interaction promotes modularity in the specification of the system. For example, consider the 2-out-of-3 problem. Since an explicit interaction is required, we need to have explicit arrows between transitions and places belonging to the memory scheduler and processes requesting memory blocks. The equivalent STOCS machine as shown in Figure 5.5 is much simpler.

3) Partial specification is difficult in Petri nets.

This is the major disadvantage of Petri nets compared to STOCS machines. Since there is no notion of communication between multiple Petri nets, the behavior of a system is generally specified by one big Petri net. This makes specification of the system difficult to understand and write. As another consequence, the system has to be specified completely before it can be analyzed. To appreciate this, consider the example of job scheduling discussed in Section 4.1 in Chapter 3. Figure 3.11 shows a Petri net for the shop and Figure 3.12 shows a STOCS machine for it. In the STOCS model, order, operators and machines are specified separately. As a result, it is easier to add another order, machine or operator to the STOCS machine than to the Petri net.

Sometimes, the communication between various Petri nets is represented using tokens. Each Petri net is considered to have input and output places. Two processes are composed by overlapping the output of one with the input of the other. This way of communication is more suitable for asynchronous messages and cannot represent synchronous events.

4) STOCS machines have a closer correspondence with state machines.

Each unit in a STOCS machine can be thought of as a generalized finite state machine. Since the notion of state arises in many contexts, it is easier to write specifications in the STOCS model than in Petri nets. Each token in a unit roughly corresponds to the current state of a finite state machine it is simulating. Consider again the example of shop modeling. Each machine and operator has a finite number of states and is easily modeled as a finite state machine.

5) Languages accepted by STOCS have an algebraic characterization

As shown in Chapter 4, the language accepted by a STOCS machine can be characterized by a concurrent regular expression. These expressions are built of algebraic operations on strings and form a suitable basis for specifying many concurrent systems. Chapter 4 provides examples of concurrent systems which are easier to model algebraically. The duality between STOCS machines and concurrent regular expressions is therefore useful for choosing the appropriate specification for any domain.

To illustrate our arguments, consider the producer consumer problem. The solution to the problem expressed in the STOCS model and Petri nets is shown in Figure 5.6. Note the following advantages of STOCS model over Petri nets.

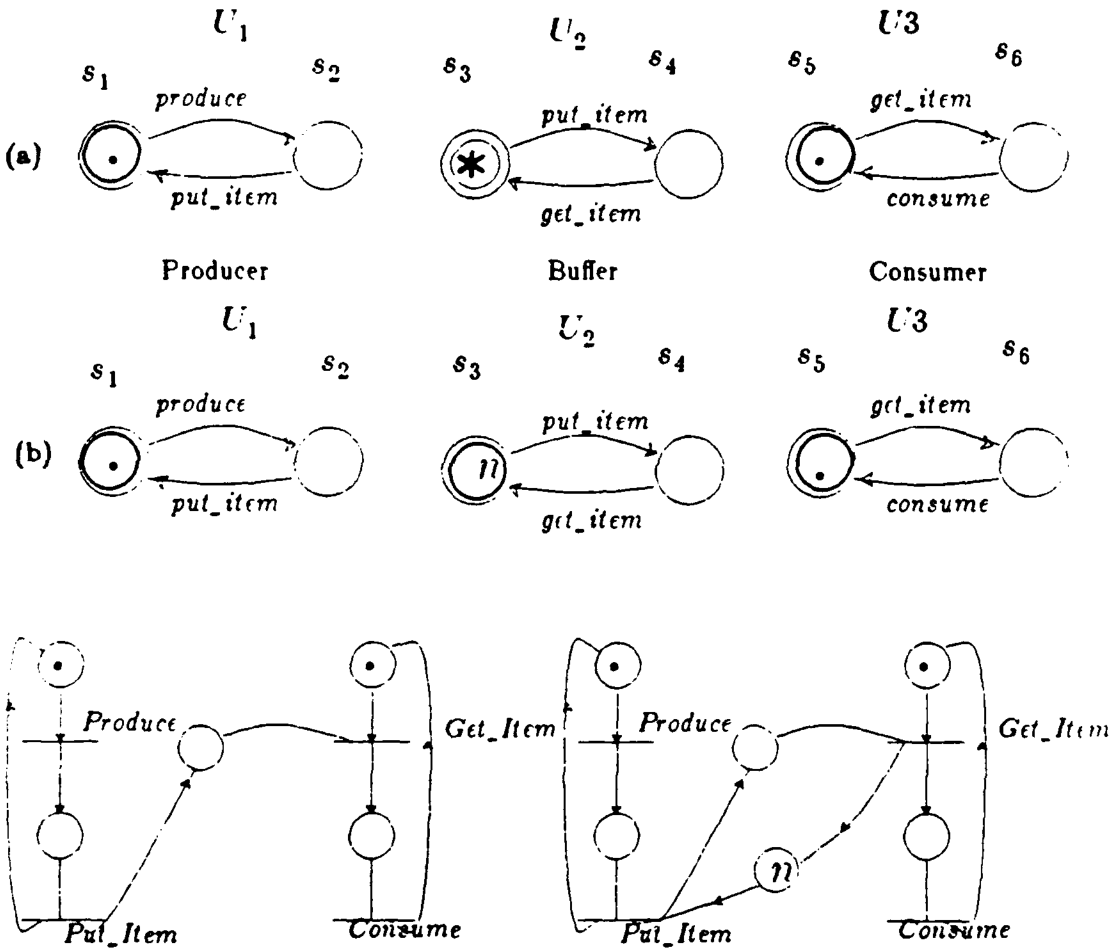


Figure 5.6: A Petri net and a STOCS machine for producer consumer problem

1) The Petri net representation consists of one single Petri net for the producer, the consumer and the buffer.

2) The unbounded version is exactly analogous to the bounded version for STOCS(* tokens instead of n). This is not true for the Petri net.

3) It is easier to specify each component of the system separately. For example, the behavior of the producer process is simply modeled as (produce put_buffer)*.

4. Comparison of Languages

A Petri Net can be defined as a four tuple (P,T,I,O) where P stands for the set of places, T stands for the set of transitions, I stands for the set of input arcs and O for the output arcs. In addition, we also define a labeling function $\sigma : T \rightarrow \Sigma$ where Σ is the alphabet of the Petri Net. We also define an initial marking μ_0 which assigns a certain number of "tokens" to places. The *function δ is a transformation function* which associates a marking and a sequence of transition firings to a new marking. Depending on the acceptance criteria four different types of languages for Petri Nets have been defined [Peterson 81]. These languages are called L,G,T and P-type languages. The acceptance criteria for a L-type language is that the Petri Net should start from the initial configuration and reach one of the predefined final configurations. In G-type languages the final configuration should cover at least one final configuration. A T-type language is accepted if the Petri Net reaches a terminal configuration i.e., a configuration where all transitions are disabled. A P-type language is a L-type language where all reachable configurations are final configurations i.e, $s=\sigma(\beta)$ is accepted if $\delta(\mu_0,\beta)$ is defined.

In addition to these classes of Petri Net languages, we define a new class of languages called F-type Languages. The definition of F-type languages is as follows. A Language L is a F-type Petri net language if there exists a Petri net structure (P,T,I,O), a free labeling of the transitions $\sigma : T \rightarrow \Sigma$, an initial marking μ_0 and a set of final states F such that L = $\{\sigma(s) \in \Sigma^* \mid s \in T^*$ and the marking $\mu = \delta(\mu_0, s)$ has no tokens in any place $\in (P-F)\}$. In this dissertation, a Petri net language would always mean an F-type language.

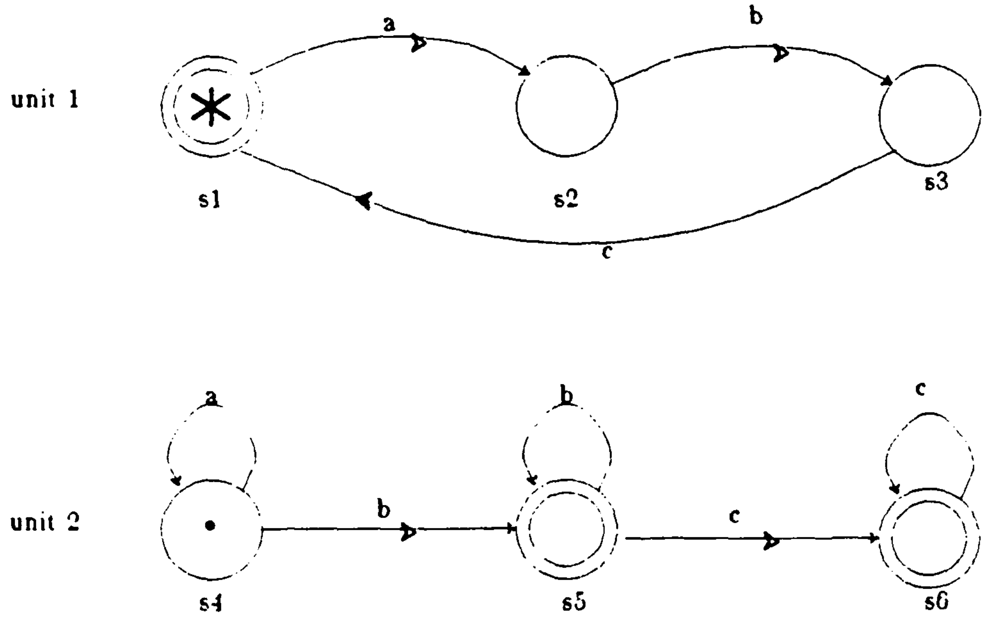
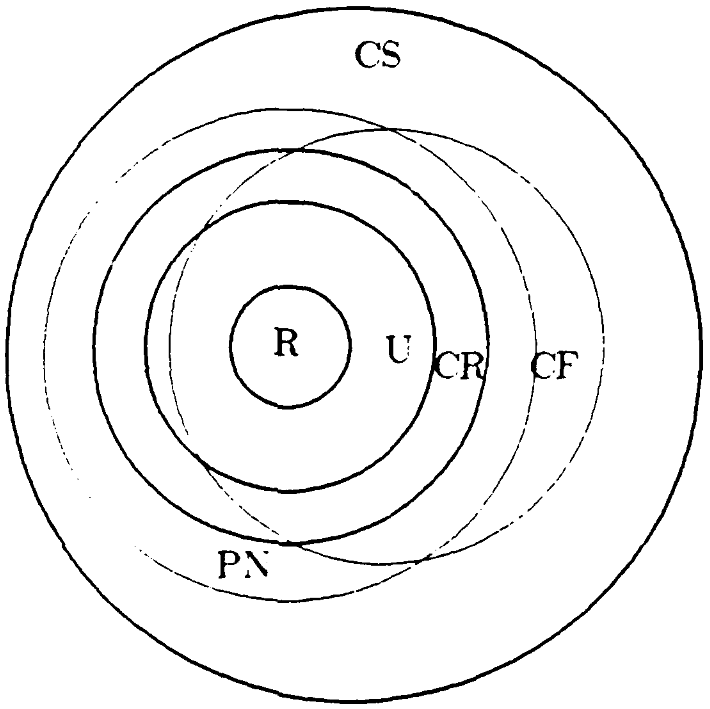


Figure 5.1: A STOCS machine accepting $a^n b^n c^n$

Theorem 5.5: All concurrent regular languages are also Petri net languages.

Proof: From the construction of Theorem 5.2. For any STOCS, the Petri net constructed has the same language as the STOCS machine.

The above result also tell us that there exist context-free languages which are not Petri net languages and therefore not concurrent regular. An example of such a language is $\{ww^R\}$ which can not be accepted by a Petri net [Peterson 81]. Figure 5.7 demonstrates that there are concurrent languages that are not context-free. The language accepted by the STOCS machine in Figure 5.7 is L= $\{a^n b^n c^n \mid n > 0\}$ which is not context free. Since the machine is deterministic, we have shown that there are DSTOCS languages that are not context free either.



CS: Context-sensitive	CR: Concurrent regular
CF: Context-free	U : Unit languages
PN: Petri net languages	R: Regular

Figure 5.8: Relation of Concurrent Regular Languages with other classes

Since concurrent regular languages are included in Petri net languages, which are properly included in context sensitive languages, we conclude that concurrent regular languages are properly included in context sensitive languages. Figure 5.8 shows the relationship of various classes of languages by a Venn diagram.

Theorem 5.4 also provides us a method of characterizing the language of ordinary Petri nets by means of algebraic expressions. Concurrent regular expressions defined so far characterize the class of languages of STOCS machines. However, the class of language of STOCS machines is a proper subset of that accepted by Petri nets. Since any *free labeled* ordinary Petri net is characterized by a concurrent regular expression, an ordinary Petri net can be described by extending CRE's with the substitution operator.

A substitution operator denoted by $<x:y>$ replaces every occurrence of x by y. For example, $ab^*c<c:a>$ is the same as ab^*a. An extended concurrent regular expression (ECRE) over Σ is defined as follows.

Definition: A unit expression is an ECRE. If A and B are ECRE's then so are $A+B, A.B, A||B, A[]B$, and $A<X:Y>$.

Theorem 5.6: The language accepted by ordinary Petri nets without ϵ is the same as that characterized by an ECRE under our acceptance condition.

Proof:

1) Petri Net => ECRE

Convert the Petri net P to a free labeled Petri net P' by assigning a substitution $<X:Y>$. By Theorem 3.2, the language of P' is the same as that of a STOCS machine S. By Theorem 4.5, the language of a STOCS machine is the same as that of a CRE C. Then

$L(P) = L(P')<Y:X> = L(CRE)<X:Y> = L(CRE<X:Y>)$

2) ECRE => Petri Net

Every unit expression can be converted to a Petri net by Theorem 3.2 and 4.5. Petri nets are closed under choice, concatenation, interleaving, substitution and synchronous composition, and therefore the result. *Q.E.D.*

5. Conclusions

In this chapter, we have shown that the reachability problem is equivalent for STOCS machines and Petri nets. This provides us the confidence in modeling capabilities of STOCS machines, because any system that can be modeled as configurations of a Petri net can equivalently be modeled by a STOCS machine. We have also shown that STOCS machines offer many advantages over Petri nets for modeling concurrent systems based on synchronous communication.

CHAPTER 6

STOCS Machines With Uncontrollable Event

1. Introduction

In this chapter, we provide an extensional theory of concurrent processes that may have *uncontrollable* events. These events are not observable by the environment and their execution depends entirely on the process. To provide the extensional theory of such processes, we retain our previous notion of equivalence. i.e. two concurrent systems are equivalent if and only if the observer cannot distinguish between them by supplying an input string and observing whether they accept it. However, we now assume, that for a given string. a system may sometimes accept it and sometimes reject it. This assumption is different from that made in classical formal languge theory. which requires a machine to either always accept a string or always reject it. In concurrent systems, and more generally in nature. there exists an uncertainty in execution which implies that any given string may sometimes be accepted and sometimes rejected. Example of such scenarios are as follows:

(1) **Timeout**: A process may change its internal state on timeout and events which were valid earlier may not be so any more. For example, consider an elevator in a building with three floors. If a passanger enters the elevator on the second floor, he can press the button for either floor 1, or floor 3. However. if he does not do so within some time, the elevator does a timeout and starts going down. In this state it will satisfy request only for floor 1. The behavior of

the elevator is shown in Figure 6.1(a). Consider an alternative design of the elevator which does not timeout. This elevator E' is shown in Figure 6.1(b). By the semantics of classical finite state machine theory, the finite state machines corresponding to two designs of of the elevators are equivalent because they accept the same language. According to our semantics, these machines will be treated different because the first one may reject a request for the third floor, whereas the second one cannot.

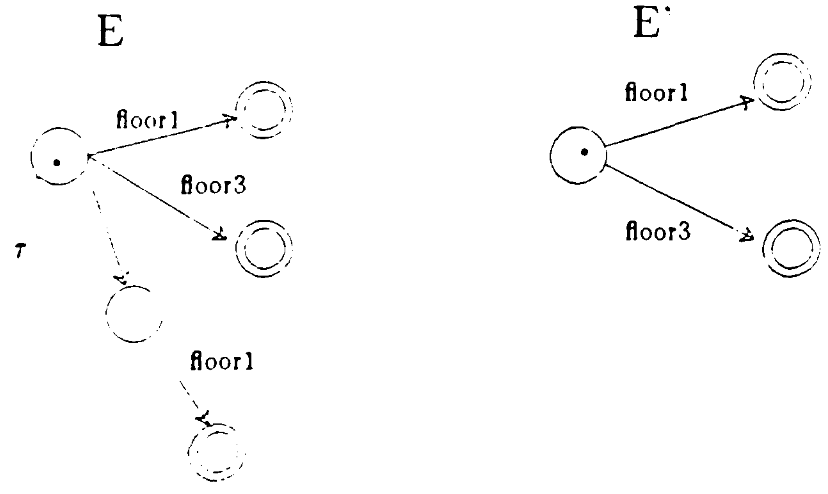


Figure 6.1: Two Different Elevators

(2) **Random**: A process may use randomness and change its state on its own. For example, consider a double-or-nothing game. Assume that you toss a fair coin, and depending on its result, you win or lose. Consider a second game in which you know how to toss the coin to get the desired side. In this game you have the choice of winning or losing. Both situations are shown in the Figure 6.2. The classical finite state machine semantics do not differentiate between the two cases as the language acceptable in either case, is {**toss.head**, **toss.tail**). With our semantics, the first machine M_1 has randomness and may accept or reject both head and tail depending on which τ the machine

takes. In contrast, M_2 does not reject any of them (i.e., the coin's outcome is dictated by its environment).

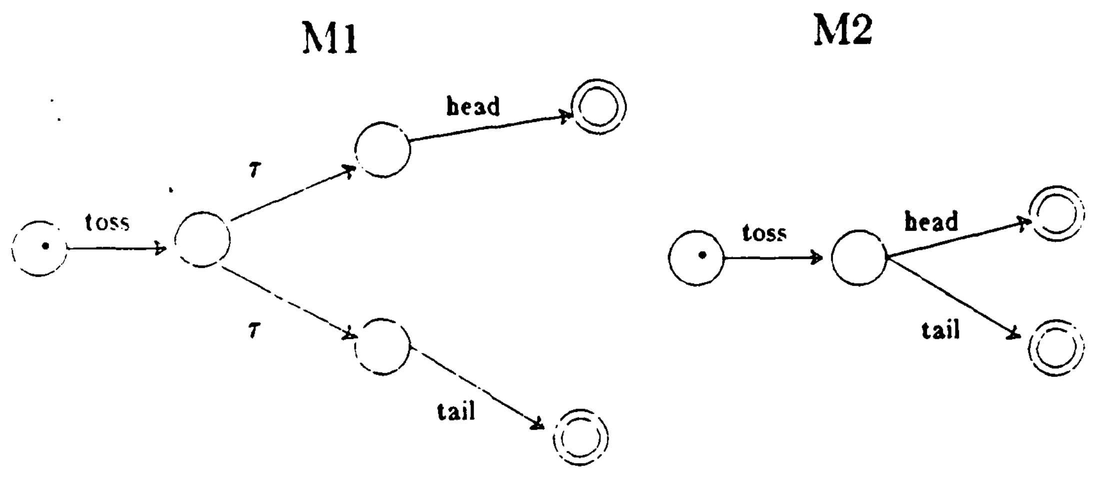


Figure 6.2: Controllable and Uncontrollable Toss

(3) **Internal Faults**: The machine may make internal state change due to a fault. A fault could be any unanticipated event for a process - such as an error in disk writing, opening a file, and termination of the communicating process. The modeling of such situations requires the use of uncontrollable events.

(4) **Hidden Events**: For abstraction purposes, we may chose to call certain events in a process, internal. These events can be executed by the machine on its own will. Consider for example, a passanger who chooses to take either a train or a bus based on some complex reasoning. For the analysis this reasoning may be irrelevant and therefore we will represent the external behavior as shown in the Figure 6.3.

With the motivation provided above, we allow a machine to take some unobservable actions. These actions, however, must terminate, and we consider any machine that can engage in unobservable actions in an unbounded manner an invalid machine. A machine, when offered a choice of events can

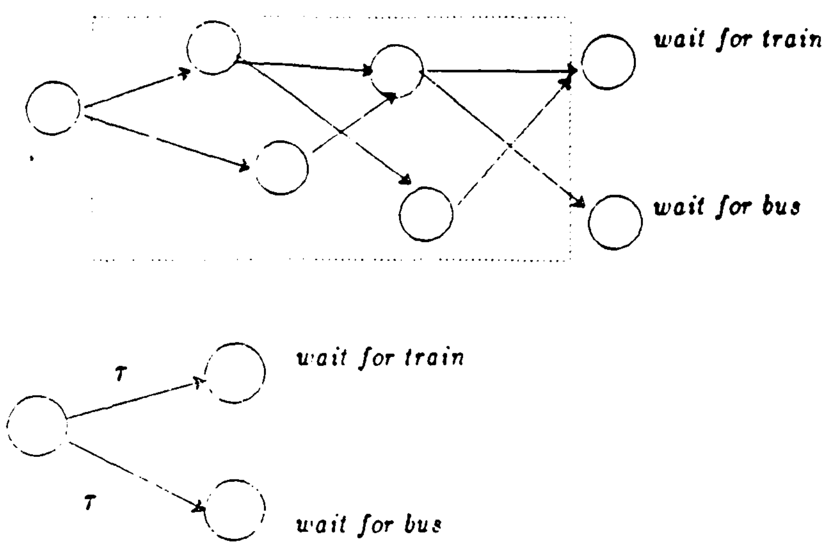


Figure 6.3: Use of τ Symbols for Abstraction

reject the experiment if and only if it cannot participate in any event or take any internal action.

This chapter is organized as follows. Section 2 compares our framework for hidden events with that proposed by Milner and Hoare[Milner 80, Hoare 85]. Section 3 describes the semantics of uncontrollable STOCS machines. Section 4 describes the semantics of uncontrollable concurrent regular processes. Section 5 proves that the equivalence of the STOCS model and CRE holds even when the uncontrollable actions are incorporated in the system.

2. Related Work

Uncontrollable actions have also been modeled and analyzed by CCS and CSP. Our work differs from them in following:

Automata Theoretic Equivalence

CCS and CSP are algebraic systems and they do not have any equivalent notions in automata theory. Whereas our theory has useful operators such as Kleene closure required to define finite state processes, such operators are missing in both CCS and CSP.

Prefix Property

Both CCS and CSP satisfy the prefix property, that is, if a sequence of event is acceptable then all its prefixes are also acceptable to the system. Our framework is more general as it can model situations in which the prefix property may not hold. If we do want the prefix property, then we can easily simulate it by considering all places final.

Separation of Specification and Operational Non-determinism

Classical non-deterministic finite state machines provide non-determinism during the specification of the system. This leads to a compact description of the system. During the operation, there is only *oracular* non-determinism as it is assumed that the machines make a correct guess at each choice presented. Such non-determinism also arises due to ϵ arcs in finite state machines, which has the semantics that the finite state machine takes that arc if it can lead to acceptance of the string.

On the other hand, CCS and CSP provide demonic non-determinism at the operation level, which we call uncontrollability. The user cannot specify that a process must take the choice which is the best for that string. If there are multiple choices that are compatible with the environment the process can make any of the choices. τ represents an internal action and the process can take this action whenever it desires so. Even though this approach can model situations not possible in classical finite state machines (as shown in Section 6.1), it loses the compactness of the non-deterministic finite state machines. In our theory, we have both kinds of non-determinism. τ provides operational non-determinism and we call it uncontrollability. ϵ and multiple symbols on a single state provide non-determinism at the specification level.

3. Uncontrollable STOCS Machines (STOCS with τ)

To describe our model extended with hidden operations, we first describe the valid syntactical structures expressed in our formalism and then describe their denotational semantics.

3.1. Syntax

A USTOCS (Uncontrollable STOCS) machine M is a set of units (U_1, U_2, U_n). Each unit is a five tuple i.e. $U_i = (P_i, C_i, \Sigma_i, \delta_i, F_i)$ where:

- P_i is a finite set of *places*
- C_i is an initial *configuration* which is a function from the set of places to nonnegative integers N and a special symbol '*'. i.e.,$C_i : P_i \rightarrow (N \cup \{*\})$.
- Σ_i is a finite set of *handshake* labels
- $\delta_i \subseteq P_i \times \Sigma_i \cup \{\epsilon, \tau\} \times P_i$,
- F_i is a set of final places, $F_i \subseteq P_i$.

The above definition is the same as that of a STOCS machine except that an arc can be labeled by any of the events in the alphabet set or by an ϵ or a τ symbol. We describe these symbols next.

ϵ: This symbol has the same semantics as in classical automata theory. It provides non-determinism in the specification of a system. From an operational view, we say that a machine consults an oracle if it has multiple choice.

τ: This symbol stands for some internal action by a machine. The external environment has no control over this action. This symbol can be used by a machine to make an internal choice. The internal action is treated like an algorithm which must terminate. This symbol provides non-determinism during the operation of the machine.

For simplicity, we chose to disallow the case when the machine does not terminate on the given input by going through a loop of internal actions. Since in real life, we would not like to have such machines we consider only those machines syntactically valid which do not have a loop of internal actions. In other words, we do not allow any unit that has a cycle consisting entirely of τ symbols. Figure 6.4 shows such a unit.

This restriction is for clarity and can be easily removed by providing semantics to the processes that can loop on internal actions. To add such semantics, it would be necessary to add the notion of divergence set [Hoare 85] which is the set of strings that can lead the machine into a nonterminating computation.

With this additional constraint on the validity of a system, we can assume that given a string a machine always terminates.

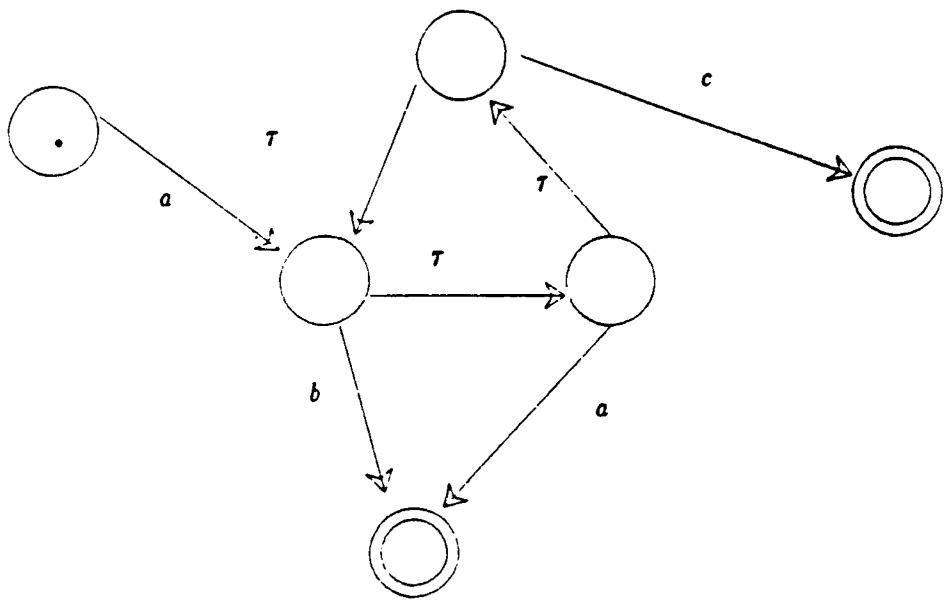


Figure 6.4: A Syntactically Invalid Machine

3.2. Semantics

In our earlier discussion, we had assumed that STOCS machines did not have τ symbols. The semantics of such a machine was defined as a tuple of its alphabet and its acceptance language. As we saw in our previous example, these two sets may not characterize a USTOCS machine completely. The semantics of a STOCS machine with τ is defined as the triple $(\Sigma, maxL, minL)$ where:

(1) Σ (Symbol Set): It is the set of symbols for the machine. For example, the set of symbols for the machine M1 is {floor1, floor3}.

(2) maxL (Optimistic Acceptance Set): It is the set of strings that can be accepted by the machine. This is the conventional trace defined for a non-deterministic finite state machine. Thus a string s is acceptable if and only if there exists a way of accepting the string s. At each node the machine chooses

any of the choices afforded to it. A τ is identical to an ϵ for this set. For example, the optimistic acceptance set for both machines E and E' is {floor1, floor3}.

(3) minL (Pessimistic Acceptance Set): It is the set of strings which are always accepted. We assume that the machine can take an internal action whenever it desires so. Therefore, at each stage of decision, the machine can either make an oracular guess or take an internal action. If there are multiple internal actions (multiple outgoing τ arcs), the machine may choose any of them.

For example, the minimum acceptance set for E is {floor1} while for E' it is {floor1, floor3}.

Example

The semantics function S for M_1 in Figure 6.1 is

$S[|E|] = \{(floor1, floor3), (floor1, floor3), (floor1)\}$

and for E', it is

$S[|E'|] = \{(floor1, floor3), (floor1, floor3), (floor1, floor3)\}$

Similarly, the semantics functions for M_1 and M_2 in Figure 6.2 are

$S[|M_1|] = \{(toss, head, tail), (toss.head, toss.tail), ()\}$

$S[|M2|] = \{(toss, head, tail), (toss.head, toss.tail), (toss.head, toss.tail)\}$

We next show that the minimum acceptance set is the same as the set of strings that must be accepted by the machine if at each of the node machine choses an internal action if possible.

Definition: The *tau-acceptance* set of a USTOCS machine is the set of strings traced by tokens such that if a token reaches a place with one or more out-going arcs labeled τ then it can only take one of them.

Theorem 6.1: The minimum acceptance set of a USTOCS machine is the same as the tau-acceptance set.

Proof: By the definition of minimum acceptance set, minL $\subseteq$ tau-acceptance set. Let a string belong to tau-acceptance set. We will show that this string cannot be rejected. At each point during the simulation of machine, it may either make an oracular guess or or take a τ action. Since on taking τ, the string gets accepted, the machine always take a τ action. Therefore, the string also belongs to the min-acceptance set. *Q.E.D.*

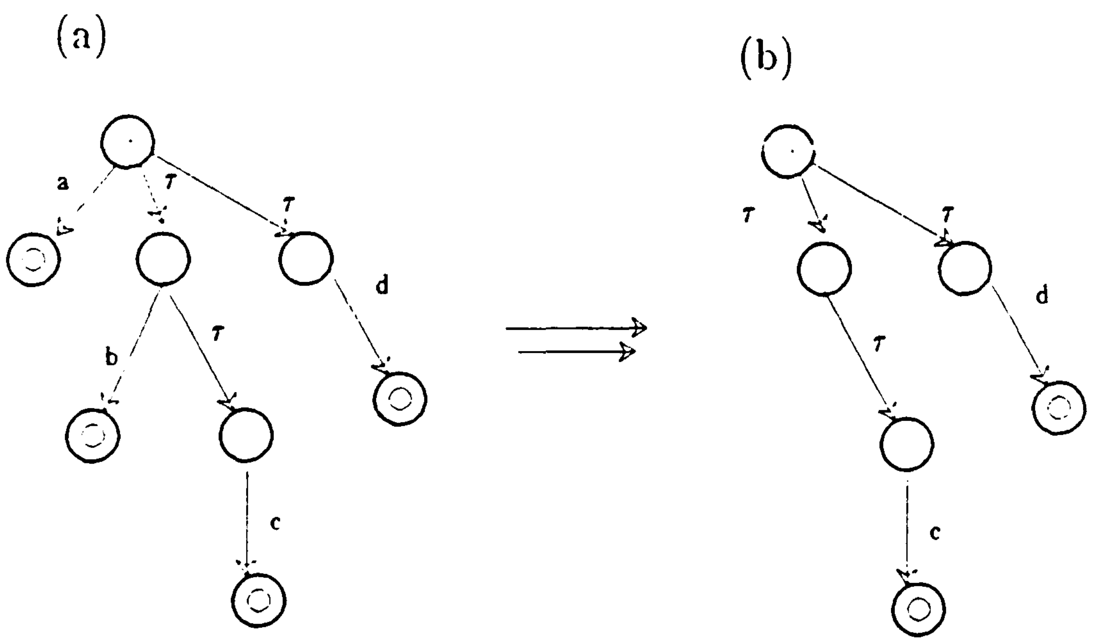


Figure 6.5: Equivalent Structures for minL

Theorem 6.1 provides us an easy way to calculate the minL for a USTOCS machine. For all places with τ as out-going edges, we delete non-tau edges as they can never be taken for minL. The resulting USTOCS machine will have two types of places - ones with out-going arcs labeled as τ and others with out-going edges labeled with epsilon or labels from Σ. By Theorem 6.1, the minL of both machines is identical. For examples, to evaluate the minL of the machine in Figure 6.5(a), it is sufficient to evaluate the minL of the machine in Figure 6.5(b).

4. Uncontrollable CRE (UCRE)

4.1. Syntax of Uncontrollable Concurrent Regular Expressions

We add an additional operator for the semantics of τ. This operator is termed *non-deterministic or* by Hoare. We chose to call this operator uncontrollable choice and denote it by $\oplus$.

<regular> :: <symbol> | <regular>* | <regular>.<regular> |

<regular> + <regular> | <regular> $\oplus$ <regular>

<unit> :: <regular> | <unit> || <unit> | <unit>$^\alpha$

<concurrent_regular> :: <unit> |

<concurrent_regular> [] <concurrent_regular>

4.2. Semantics of Concurrent Regular Expressions

If concurrent regular expressions are to characterize USTOCS machines, their semantics must also be specified as a triple $(\Sigma, maxL, minL)$. The new operator, uncontrollable choice, is identical to ordinary choice for the purposes of maxL but different for minL. Consider the expressions (a+b) and (a $\oplus$ b). Both of them accept the language {a,b}. However it is possible that (a $\oplus$ b) not accept a or b, whereas a+b will always do so.

With the above intuition, we formally define the semantics of concurrent regular expressions as follows:

Primitive Regular Expressions:

$$S[|\epsilon|] = \{(),(\epsilon),(\epsilon)\}$$

$$S[|\tau|] = \{(),(),()\}$$

$$S[|a|] = \{(a),(a),(a)\} \qquad \forall a \in \Sigma$$

Controllable Choice:

$$S[|A+B|] = \{\Sigma_A \cup \Sigma_B, O_A \cup O_B, P_A \cup P_B\}$$

Uncontrollable Choice:

This operator is responsible for the differences between the maximum and minimum language.

$$S[|A \hat{+} B|] = \{\Sigma_A \cup \Sigma_B, O(A) \cup O(B), P(A) \cap P(B)\}$$

Concatenation:

$$S[|A.B|] = \{\Sigma_A \cup \Sigma_B, O_A . O_B, P_A . P_B\}$$

Kleene-Closure:

$$S[|A^*|] = \{\Sigma_A, O_A{}^*, P_A{}^*\}$$

Interleaving:

$$S[|A||B|] = \{\Sigma_A \cup \Sigma_B, O_A || O_B, P_A || P_B\}$$

Alpha-Closure:

$$S[|A^\alpha|] = \{\Sigma_A, O_A{}^\alpha, P_A{}^\alpha\}$$

Synchronous Composition:

$S[|A[]B|] = \{\Sigma_A \cup \Sigma_B . O_A[]O_B . P_A[]P_B\}$

Example 1: Assume that we need to model the fact that the machine must chose a but after that it may accept b as well as c.

$S[|a.(b+c)|] = \{(a,b,c),(ab,ac),(ab,ac)\}$

On the other hand, if the machine is such that after executing a, it may accept b or c, but also reject either of them. Then,

$S[|a.(b \overset{\frown}{+} c)|] = \{(a,b,c),(ab,ac),()\}$

Example 2: We now give the semantic functions of some non-trivial examples.

$S[|(ac+b) \overset{\frown}{+} (a.(c+b))|] = \{(a,b,c),(ac,b,ab),(ac)\}$

$S[|((ac+bd) \overset{\frown}{+} (bd))^{\cap}|] = \{(a,b,c,d),(ac.bd)^{\cap},(bd)^{\cap}\}$

5. Equivalence of USTOCS Machines and UCRE's

We will show in this section that USTOCS machines and UCRE's are equivalent in power.

5.1. Construction of a USTOCS machine from a UCRE

We will show that a regular expression with $\overset{\frown}{+}$ can be converted to a finite state machines with τ's. It is easy to extend this construction for a more general case of UCRE.

We first show that the maximal and minimum acceptance set of a URE are regular sets. The maximal acceptance set of a URE is obviously a regular set. The following Lemma shows that it is also true for the minimum acceptance set.

Lemma 6.1: Minimum acceptance set of a URE is a regular set.

Proof: We use induction on the structure of URE. Except $\oplus$, all operators treat the maximum acceptance set and the minimum acceptance set in an identical manner. For $\oplus$, we take the intersection of two sets. As regular sets are closed under intersection, this would result in another regular set.

To convert a URE to a UFSM, we write it as RE1$\oplus$RE2, where RE1 represents the maximal acceptance set and RE2 is the minimum acceptance set. For each one of them a state machine can be constructed. The total composite machine can be written as follows. Construct a new start state. Connect τ arcs to I and I'. The maximal set of this machine is the same as the maximum set of URE because the machine can always take transition to the first finite state machine. Similarly, the minimum acceptance set is the same as that of URE because no matter which τ the machine takes, the string in minimum set belong to both I and I'.

For example, consider the URE $(aaab^* \oplus a^*bbb)aba$. The maximal set of this URE can be written as $(aaab^*+a^*bbb)aba$ while the minimum set can be written as $(aaabbb)aba$. Therefore the UFSM corresponding to the URE is as shown in the Figure 6.6.

The above construction can lead to a large UFSM. In the above construction of machine for the minimum acceptance set, we may have to take the intersection of two finite state machines to simulate $\oplus$ operator. This may result in a UFM that are considerably bigger than the given URE. We now provide a construction that keeps the size of UFSM upto a constant factor in

URE $= (aaab^* \oplus a^*bbb)aba$.

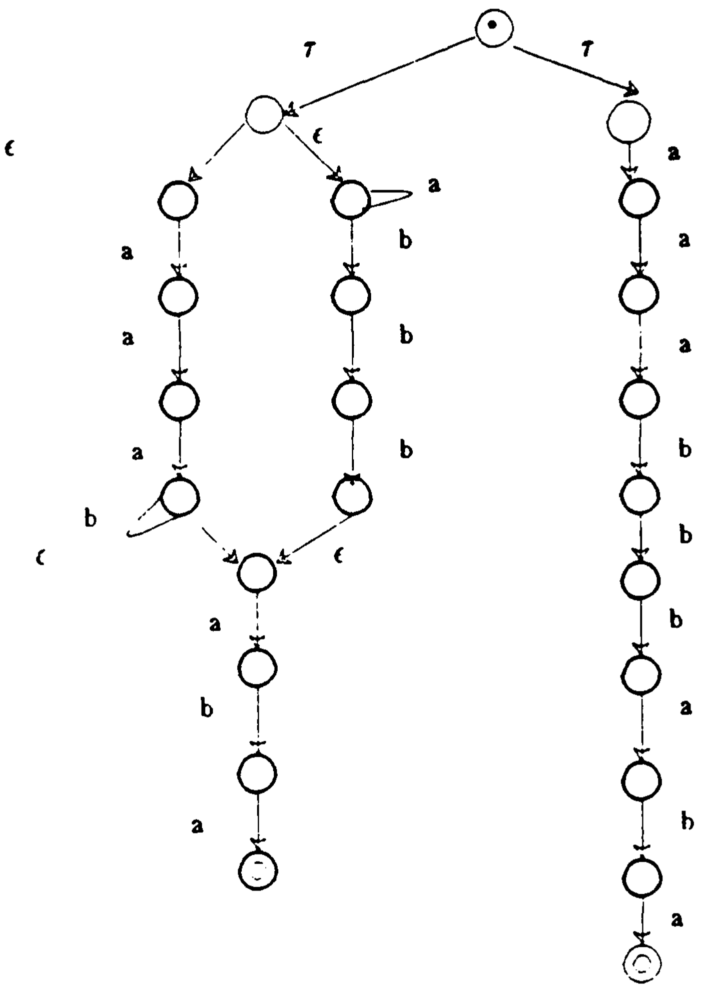


Figure 6.6: URE => UFSM

size of URE.

Theorem 6.2: There exists a linear algorithm to convert a URE into a UFSM.

Proof: For each primitive regular expression such as a and ϵ, we construct a FSM as shown in Figure 6.7. The controllable choice operator is implemented by creating a new initial state and adding ϵ arcs from the new initial state to initial states of operand machines. The uncontrollable choice operator is

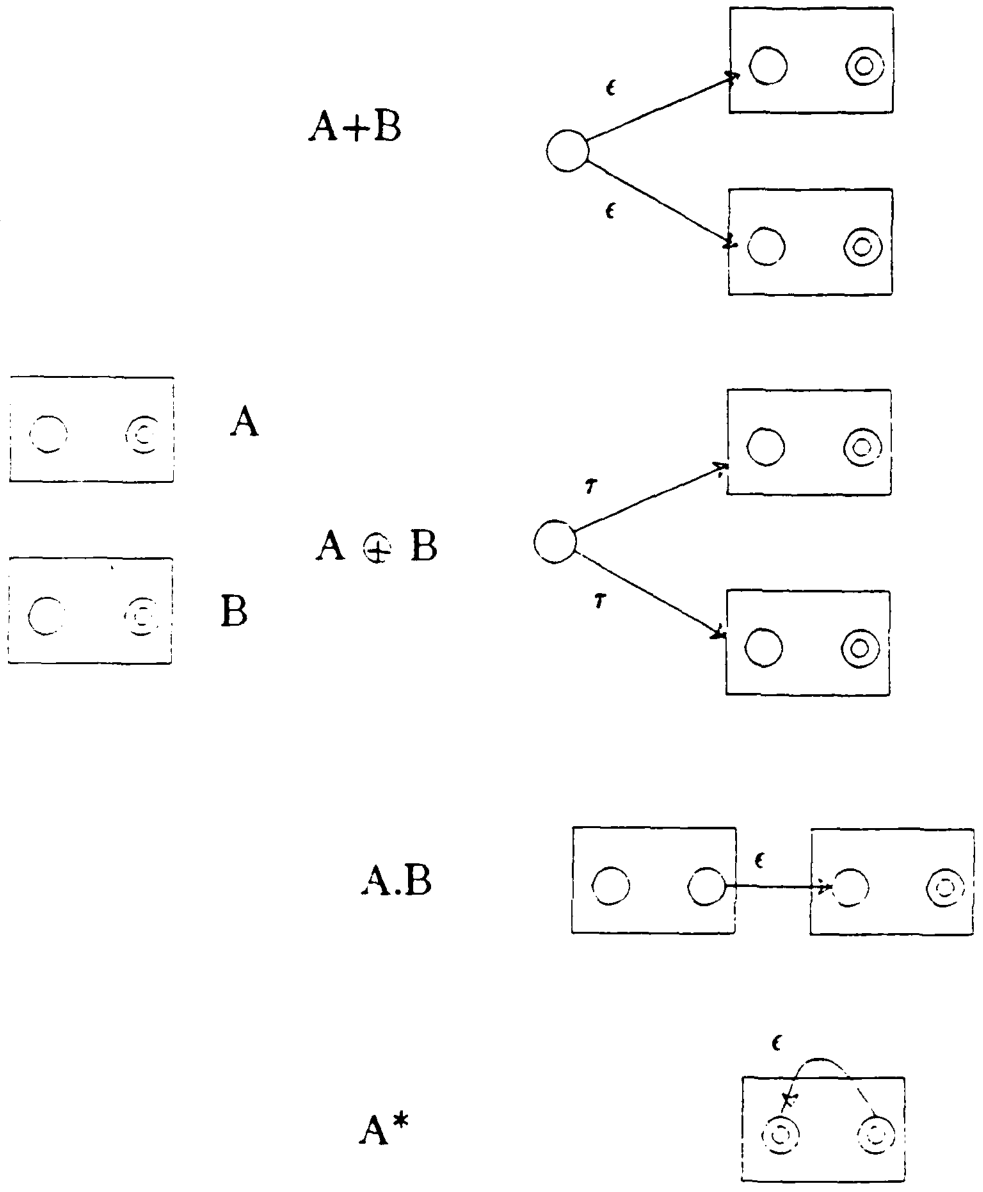


Figure 6.7: URE $=>$ UFSM. A better transformation

implemented by creating a new initial state and adding τ arcs from the new initial state to initial states of operand machines. Concatenation of A and B is implemented by means of ϵ arcs from the final states of I2 to the initial state of I1. Kleene-closure is implemented by adding ϵ arcs from all final states to the initial state and making the initial state final. *Q.E.D.* Figure 6.8 shows the application of this algorithm.

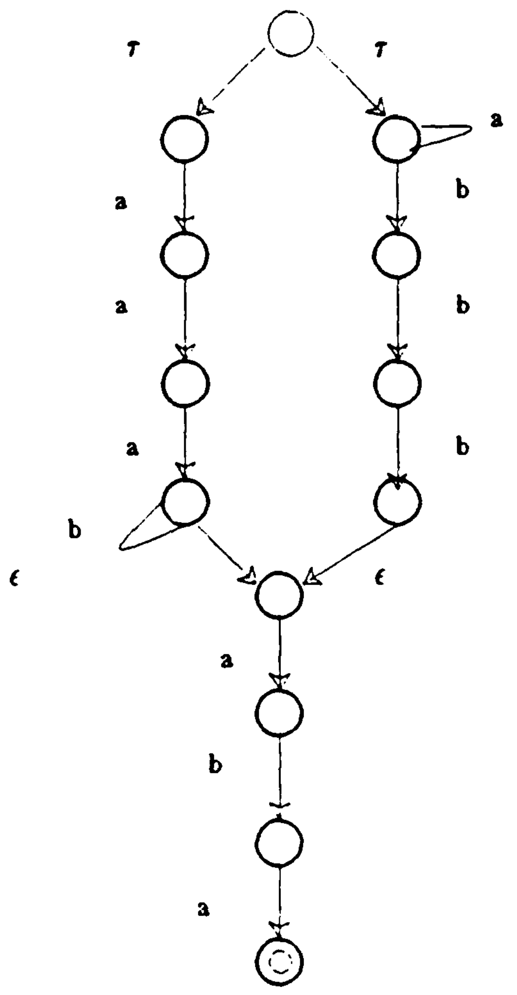


Figure 6.8: Result of more direct transformation

5.2. Construction of UCRE's from USTOCS Machines

We will show that a URE can be constructed from a UFSM. It is easy to extend the construction to USTOCS machines. We first show that a UFSM can be written as a conjunction of two finite state machines, one for maximal acceptance set and one for minimum acceptance set. Since a FSM can be converted to a RE, the URE equivalent to a UFSM is just the $\oplus$ of RE's for maximum and minimum RE's.

Theorem 6.3: Minimum acceptance set of a UFSM is a regular set.

Proof: We first use Theorem 6.1 to use tau-acceptance set of the UFSM instead of the minimum acceptance set. We show the result by reducing the number of nodes with τ-arcs. Choose any node with n out-going τ arcs. We replace this UFSM by the intersection of n machines in which this node will have only one out-going arc. Since such a node can be combined with its destination node, the total number of τ nodes is reduced by one. *Q.E.D.*

An application of Theorem 6.3 is shown in Figure 6.9.

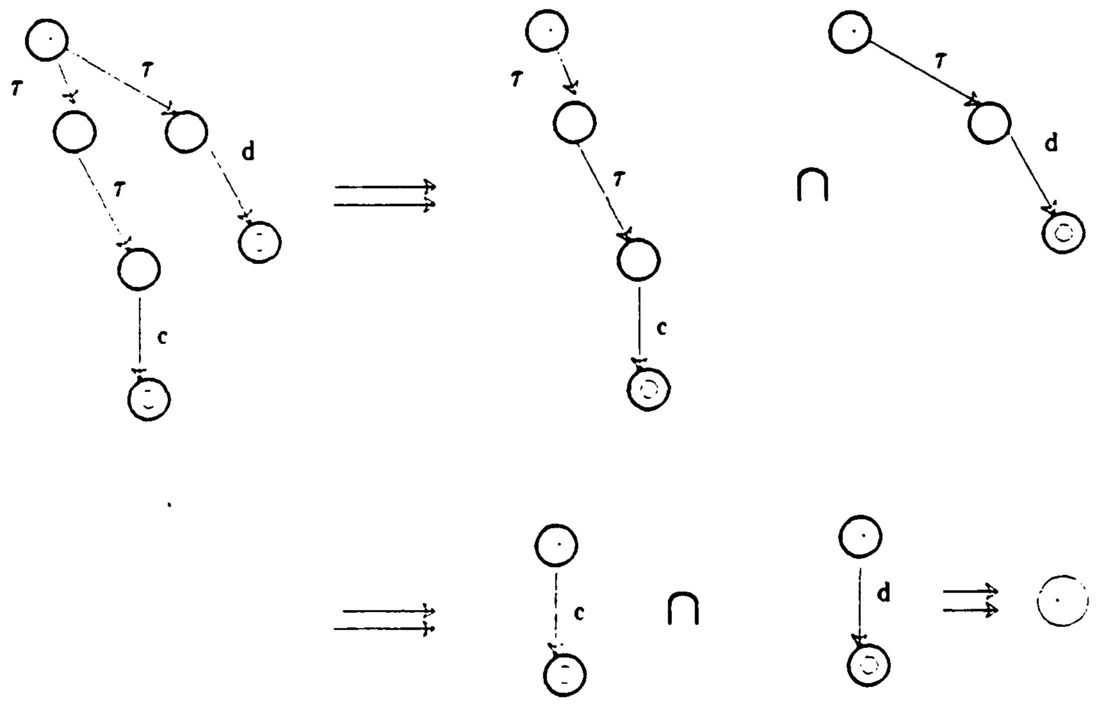


Figure 6.9: Construction of regular set for minL

6. Conclusions

This chapter shows how hidden actions can easily be incorporated in our theory of concurrent processes. We provide a unified treatment of demonic and oracular non-determinism. We believe that our approach leads to compact specification via ϵ and more general semantics via τ. We have shown in this

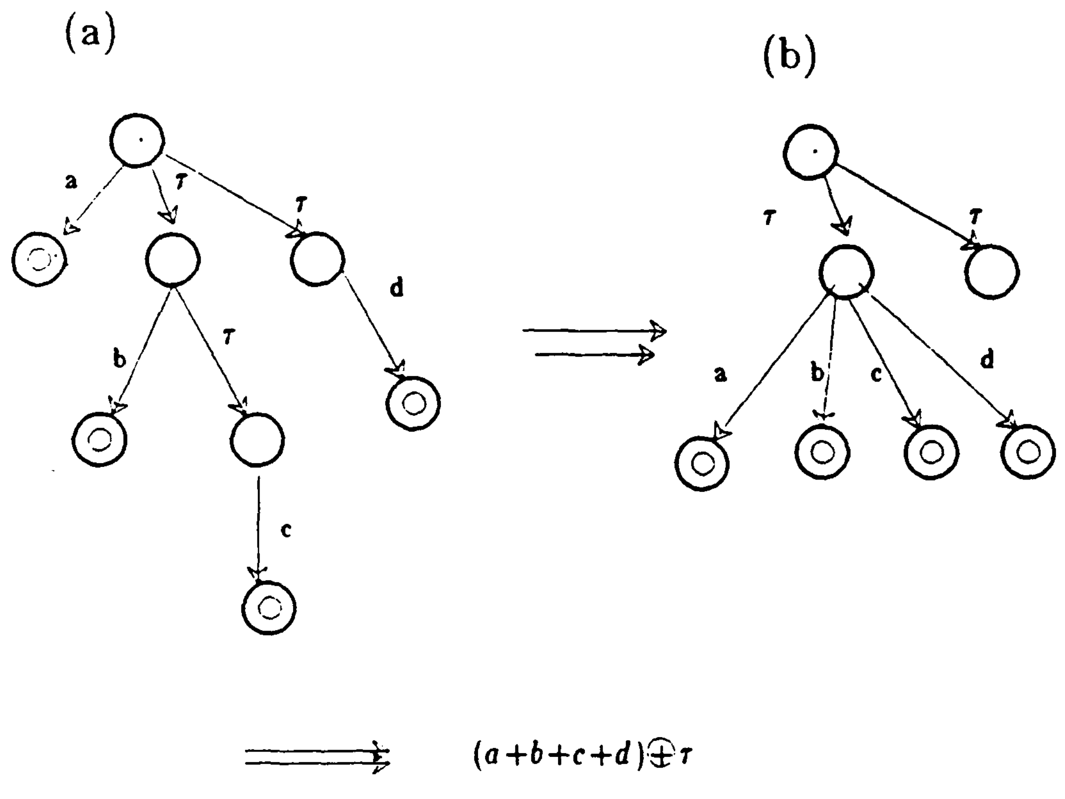


Figure 6.10: UFSM => URE

chapter that the STOCS machines with uncontrollable events are equivalent to UCRE with non-deterministic or (⊕).

CHAPTER 7

Analysis of STOCS Machines

1. Introduction

Research efforts in reasoning about programs can be divided into two groups - manual and automatic. Most researchers in distributed algorithms use manual reasoning based on the behavior of the program. Many proof systems have been developed for reasoning about safety and liveness properties [Apt 80, Hoare 85, Milner 80, Misra 81, Lamport 84]. Manual analysis is error prone and cumbersome; therefore, we will restrict our discussion to automatic analysis of distributed programs.

Automatic analysis of a concurrent system consists of computer exploration of all its possible behaviors. Many concurrent systems are based on finite state machines, making them particularly amenable to computer analysis. This approach has been used by many researchers, especially for the verification of communication protocols [Gerhart 80, Aggarwal 84, Blumer 86]. There are two main hurdles to this approach - the number of processes may not be known initially, and the number of states may be too large for exploration. For an illustration of difficulties involved in this approach consider the mutual exclusion algorithm in a ring network [Dijkstra 85, Clarke 86]. If we know the number of processes initially (say 5), then we could construct the global state graph and check for any property in the graph. However, this approach becomes infeasible if the number of processes is not known initially or is large (say 100).

There have been many efforts to contain the state explosion problem. Many researchers [Dong 83, Kurshan 85] have studied this problem in the context of automatic protocol verification, where this problem is dealt with by collapsing multiple states into a single state while preserving properties that are important for verification. These properties, however, are limited to logical properties such as liveness and safety, whereas we also include functional properties. Also, their effort cannot be used for reasoning in networks with an unknown number of identical processes. Clarke et. al. [Clarke 86] propose inductive techniques to prove properties of networks with identical finite state processes. Their approach consists of establishing a correspondence relationship between the global graph of n processes and the global graph of $n+1$ processes. They show that if the correspondence can be established, then any formula expressed in Indexed Computation Tree Logic (ICTL) † which holds in the initial state of a network with a small number of processes will hold for the network with a large number of processes. However, the step of establishing the correspondence is manual and could be difficult enough to defeat the original purpose of avoiding manual analysis. Our aim in this research is to minimize human involvement during the analysis.

In this chapter, we present algorithmic techniques based on reachability for the analysis of distributed systems. Since reachability algorithms face state space explosion, we have used two methods to cut down the state space:

† a proper subset of branch time temporal logic

exploitation of modularity and exploitation of symmetry. Exploitation of modularity deals with techniques which analyze a system in a modular manner. Exploitation of symmetry deals with symbolic and induction techniques which avoid the global state space exploration.

This chapter is organized as follows. Section 2 discusses exploitation of modularity in analyzing distributed systems. Section 3 discusses exploitation of symmetry for analysis of systems expressed in the STOCS model. Section 4 makes concluding observations.

2. Exploitation of Modularity

We exploit modularity by means of the *projection analysis method*. This method studies appropriate parts of the system to make assertions about the global behavior. Projection techniques can be useful for analyzing the safety properties of concurrent systems. They cut down the global state space by exploring only the relevant parts of the specification. It is for the user to decide which parts of the specifications are relevant.

Exploitation of modularity is easier in the STOCS model than in Petri nets which are often analyzed for the reachable configurations. The ease of analysis comes from the following reasons. First, the easier specification of partial system in the STOCS model leads to techniques for analysis of partial systems and their use for assertions about the global behavior. Second, the STOCS model has extra information about which place belongs to which process (unit assignment) and therefore it can exploit the notion of a process which is missing in Petri nets. Third, all the unboundedness of the STOCS model is explicit and confined to *-states which makes the analysis simpler.

Safety concerns are generally phrased as "the system must never reach the bad state". Example of bad states are: "two processes are in the critical region", "there is no token in the token-ring" and "there are more men than women in the ballroom". More formally, a safety property of a STOCS machine M is a logical statement of the form: Configuration C does not belong to the set of all reachable configurations. Alternatively it could be based on the sequence of computation and may assert that: the string of computation S does not belong to the language L of the machine. A system is considered safe if it satisfies all its safety properties. Figure 7.1 shows that for a safety property it may be sufficient to analyze only the relevant units. This reduces not only the number of reachable states, but also the complexity of analysis if units do not have a *-place. These units corresponds to S-invariants of the Petri net.

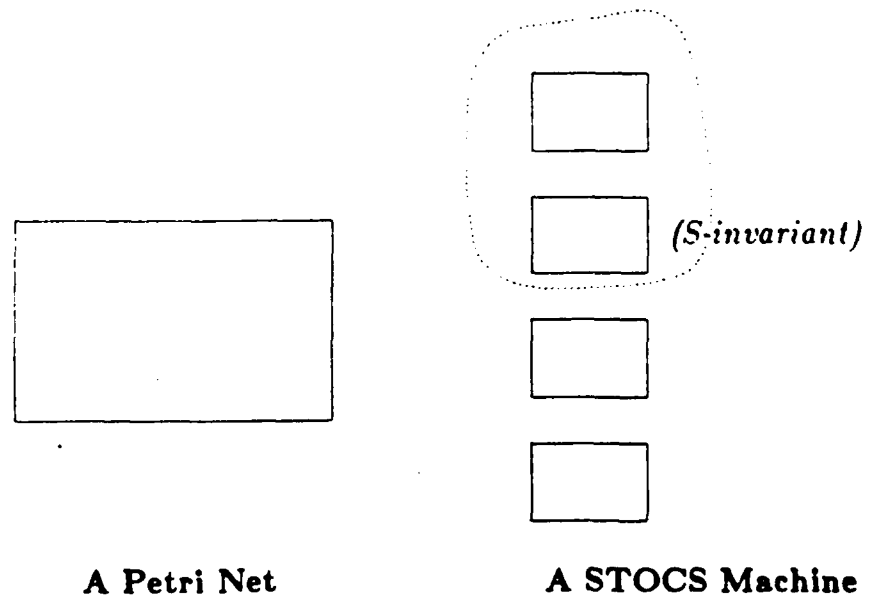


Figure 7.1: Modular Analysis of STOCS Machines

2.1. Reachable Configurations

We can do a reachability analysis for the STOCS model with the advantage of exploring just a suitable projection of the system. The analysis of reachable configurations of a subset of a system is based on the observation that if a process cannot reach a configuration assuming the absence of some units then it cannot reach that configuration in their presence. Before we state our theorem, we need the definition of projection. The *projection* of a configuration over a set S is defined as the configuration of tokens in states belonging to S. Let R(M,C) represent the set of all configurations of a STOCS machine M reachable from the initial configuration C.

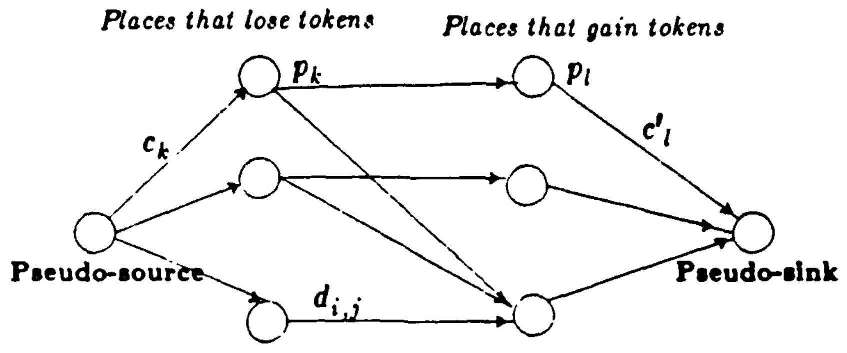
Theorem 7.1: Let M_1 and M_2 be two STOCS machines. Let C_1 and C_2 be initial configurations of both STOCS machines respectively. Then

$Proj_{M_1}(R(M_1[]M_2,(C_1,C_2)) \subseteq R(M_1,C_1)$

Proof: From definitions of projection, execution and composition. *Q.E.D.*

This theorem, although simple, has powerful applications. Using the theorem, it suffices to prove that a certain configuration is not reachable in a partial system to prove that it is not reachable in the total system. We next show that the reachability question may be considerably simpler to answer for a partial system. Theorem 7.2 gives a simple algorithm using max-flow techniques to answer any reachability question on a STOCS machine consisting of a single unit.

Theorem 7.2: Let a STOCS machine consist of a single unit with n states. There exists an algorithm which given any initial and final configuration answers the reachability question in $O(n^3)$ time (independent of the



c_k= Number of tokens lost by ith place
c'_l= Number of tokens gained by ith place
$d_{i,j}$= ∞ if a path exists otherwise 0

Figure 7.2: Construction of the Max-flow Graph

configurations themselves).

Proof

From Lemma 4.2 any unit with multiple *-places can be converted to a unit U with a single *-place by merging all the *-places. We construct a max-flow graph with n+2 nodes(Figure 7.2). We have one node for each place in the STOCS machine. We divide up the places into two sets - places which gain tokens (G) and places which lose tokens (L). If the overall final configuration has more tokens than the initial configuration, we add the *-place to L, otherwise we add the *-place to G. We connect each of the places in L to a pseudo source with an arc of capacity equal to the number of tokens they lose. Similarly, we connect each of the places in G to a pseudo sink with an arc of capacity equal to the number of tokens they gain.

We also connect two nodes with an arc of infinite capacity if there is a path between the corresponding places in the STOCS machine. We next show that the final configuration is reachable if and only if the maxflow in the graph is equal to the maximum of the number of tokens in the initial and the final configuration.

If there exists a maxflow with the desired value then all the edges with the finite capacity must get saturated. This implies that there exists a way such that (1) Places connected to the source lose desired number of tokens (2) Places connected to the sink gain desired number of tokens (3) The movement of tokens respect the reachability in the graph. These three facts together imply that the final configuration is reachable.

To see the converse assume that the final configuration is reachable. Reachability must respect the reachability conditions in the graph and therefore the change in configuration can be simulated in the graph. This will make the graph saturated implying the desired condition. *Q.E.D.*

For example, consider the buffer process with size n in the producer consumer problem. The number of tokens in the place p_3 represents the number of empty buffers and the number of tokens in the place p_4 represents the number of filled buffers. Since the number of tokens in a *-place free unit is constant, we conclude that the number of filled buffers can never exceed n.

The previous result gives us an efficient algorithm to answer any reachability question for a single unit. These units are even allowed to have *-places. The simplicity in analysis comes from the fact that the tokens within a single unit are not constrained and can move freely. The problem is more difficult with the presence of additional units. In this section we show that it is NP-complete to analyze the a STOCS machine with multiple units for reachability. We impose these additional restrictions to the structure of STOCS machine.

(1) There are no *-place.

(2) Each of the unit is acyclic.

Theorem 7.3: [Kanellakis 85] The following problem is NP-complete.

Instance: A STOCS machine S=(U1,U2), such that no unit has *-place.

Question: Is configuration C reachable?

Proof: This proof is adapted form [Kanellakis 85].

It is in NP.

This is proved by providing the steps which lead to the configuration. Since the machine is acyclic, the number of steps are polynomial in its size.

Reduction from 3-SAT.

Instance: A set U of variables, collection C of clauses over U.

Question: Is there a satisfying truth assignment?

For each variable, we make a process as shown in the Figure 7.3 and for the collection of clauses we make another unit as shown in Figure 7.3. The initial configuration is shown in the Figure. The boolean formula is satisfiable if and only if the STOCS machine can reach a configuration in which all tokens are in the last state of their units. Each component in the unit U_1, decides the value of a variable. *Q.E.D.*

The above result indicates that in general one may have to take the cross product of all the structures traced by tokens.

$(x1 \vee \overline{x2} \vee \overline{x3}) \wedge (x1 \vee x2 \vee \overline{x3}) \wedge (\overline{x1} \vee x2 \vee x3)$

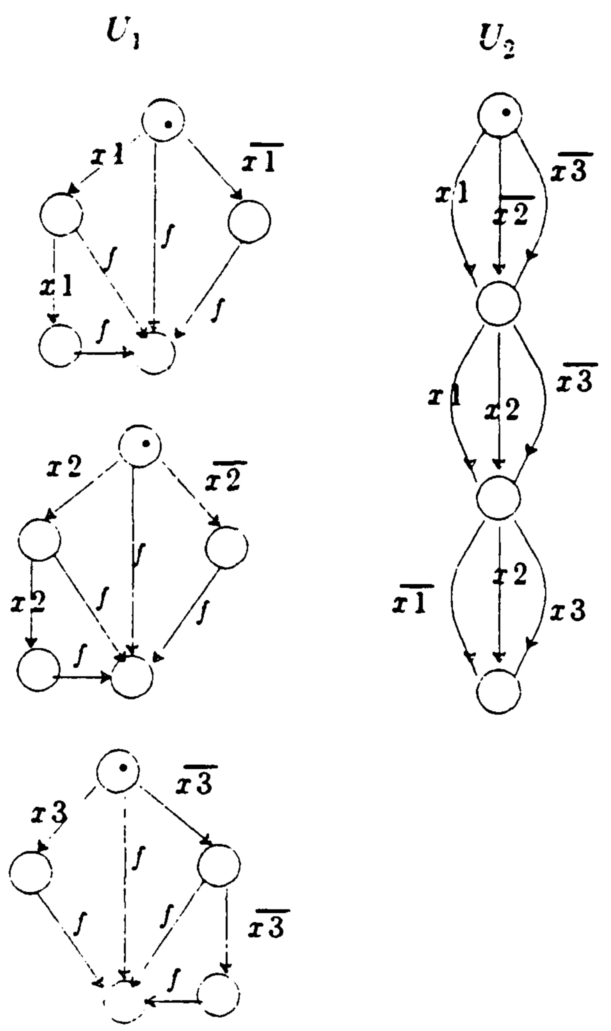


Figure 7.3: Reduction of 3-SAT to reachability in acyclic bounded STOCS machine

2.2. Language of the STOCS Machine

So far, we have been interested in reachable configurations. Other interesting questions can be posed in terms of the language of a given machine. The analysis of the language of a STOCS machine is based on a crucial observation - if a process cannot make a transition assuming the absence of some units then it cannot do so in their presence. This observation is formalized in Theorem 7.4 which states that the language of composition of two STOCS machines restricted to the alphabet of both STOCS machines is a subset of the intersection of their languages. More formally,

Theorem 7.4: Let M_1, and M_2 be two STOCS machines with H_1 and H_2 as their handshake sets. Let $H = H_1 \cap H_2$. Then

$L(M_1 [] M_2)/H \subseteq L(M_1)/H \cap L(M_2)/H$

Proof: From the definitions of composition, execution and restriction. $Q.E.D.$

For example, consider the producer consumer problem with an infinite buffer (Figure 3.2). The language of the buffer process is the set of strings consisting of symbols *put_item* and *get_item* such that the number of *put_item* is greater than or equal to the number of *get_item*. Once we know the language of the buffer, we conclude by Theorem 7.4 that no matter what other components exist in the system, this specification must hold. For the mutual exclusion problem (Figure 3.1), the language of the synchronizer process is $<reqrel>^*$. With this we are guaranteed that no matter what other processes do, there cannot be two consecutive *reqs*. Similarly, the chocolate vending machine owner in the vending machine example can satisfy himself that no customer can cheat him by analyzing the language of the vending machine. The language of the vending machine consists of all those strings such that the number of coins is greater than or equal to the number of chocolates delivered. Note that it would be harder to prove such things for Petri nets.

3. Exploitation of Symmetry

Exploitation of symmetry is facilitated by the structure of the STOCS model because tokens make it easier to model systems with many identical processes. As most distributed systems have one or more identical sets of processes, these techniques have wide applicability, especially to the networks with the *star, broadcast* or the *ring* topology (see Figure 7.4).

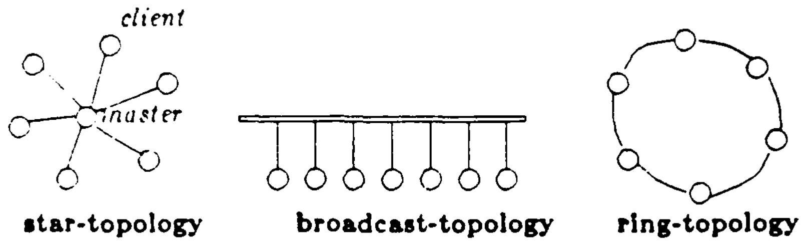


Figure 7.4: Virtual Topology for Communication

A *star topology* consists of a server (master) process and a set of identical client (slave) processes. Client processes interact only with the server process forming a star topology. This topology is common in centralized systems and various network servers such as name server and printer server. A *broadcast topology* consists of a set of identical processes connected to a broadcast medium such as an ethernet. We assume that messages are always broadcast and must be received by all the processes. An example of such a system is a set of identical readers and writers connected to a ethernet. A *ring topology* consists of a set of identical processes communicating in a circular fashion. Each process has two neighbors and all messages originating at the process must go through one of the neighbors. This topology of processes is common for local area networks with token ring such as the Cambridge Ring Network [Needham 82].

All of the above networks show symmetry and it is desirable to have methods to reduce the global state space by exploiting this symmetry. We propose symbolic and induction methods in this chapter (see Figure 7.5).

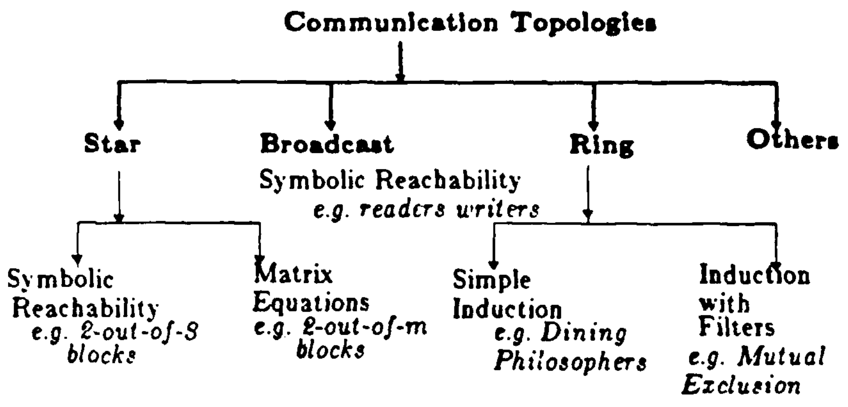


Figure 7.5: Analysis of Various Topologies

3.1. Star Topology

The symbolic analysis method expresses the global state in terms of symbols instead of computing the actual global state. These symbols are then manipulated to compute other reachable global states. A symbol could stand for any unspecified component of the system, such as the number of processes. With this method, one symbolic state represents multiple computed states, thus reducing the state space substantially.

One of the advantages of the notion of *token* in STOCS is that it can represent a process; therefore, multiple identical processes are represented by multiple tokens in some state. If the number of processes is large or is unknown initially, we may use a symbol (say n) in a state to represent the unknown number of processes. Now we do the rest of the analysis in terms of these symbols. We use symbolic analysis for networks with either a star or a broadcast topology. A star topology is shown in Figure 7.4. A STOCS representation of such a network would generally have two units - one for the master process and one for multiple slaves. The multiplicity of slaves is represented by presence of multiple tokens in some state. We will use two methods to analyze STOCS with star topology - symbolic reachability and matrix equations.

Symbolic Reachability

Symbolic reachability of a STOCS machine is done by constructing the *reachability graph* of its configurations. A reachability graph is a directed graph with each node representing a marking and a directed edge from one marking (say C_1) to another (say C_2) if there is a handshake that takes the STOCS machine from marking C_1 to C_2. We allow coordinates of a marking to be symbolic. As an example of symbolic analysis consider the 2-out-of-3 problem.

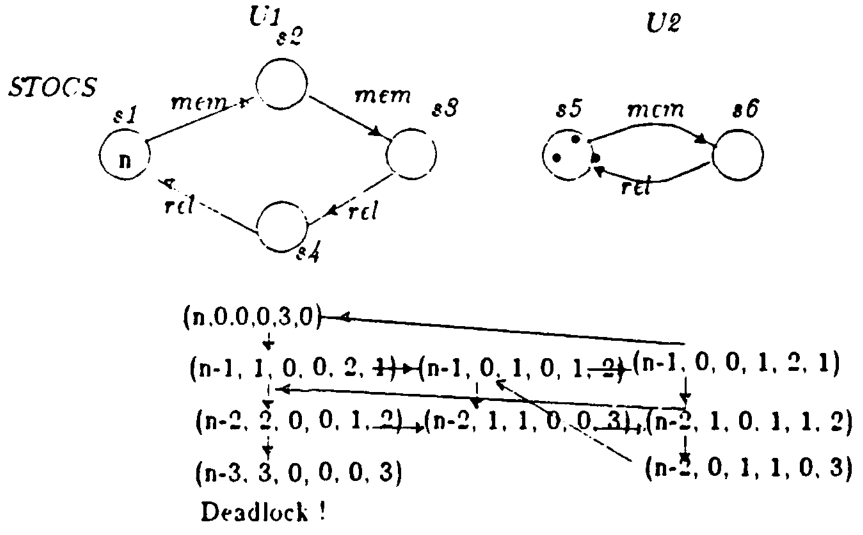


Figure 7.6: Example of symbolic analysis of STOCS

The 2-out-of-3 problem is a good abstraction of many resource contention problems. Assume that a memory scheduler has three memory blocks and that any process requires two memory blocks to execute. A non-preemptive procedure for such a system with n processes is given in Figure 7.6. We place n tokens in the state s_1 to signify n processes and three tokens in the state s_5 to signify availability of three memory blocks. To analyze the solution, we draw a reachability graph of its configurations. The initial configuration is (n,0,0,0,3,0). With this configuration only a *mem* handshake can take place, resulting in the configuration (n-1,1,0,0,2,1) which is explored next. This procedure is continued until all nodes in the graph have been explored. Following it in our example, we find that a deadlock exists if the number of processes is greater than or equal to 3 (see Figure 7.6).

With the brute force method of taking the cross product of all possible states of all processes, there would be 4^{25} states for a system with 25 processes, in contrast to 9 states that need to be explored if the symmetry is exploited. The chief disadvantage of this method is that the reachability graph may not be finite. ω-notation, first introduced by [Karp 68], can be used to make the graph finite but due to the loss of information it can only solve the coverability problem[Peterson 81]. As this method is independent of the issues that arise due to the symbolic nature of coordinates, we do not discuss this method here and refer interested readers to [Peterson 81].

Matrix Equations

Symbolic reachability can sometimes run into problems when the reachability graph is dependent on the value of symbols. Matrix equations, another approach to analyzing STOCS, are as easy to analyze with symbols as without them. We will use a variation of the example of a memory server to illustrate the analysis with matrix equations. We will assume that the memory server has *m* blocks (instead of 3) and that it uses a preemptive algorithm to grant requests (see Figure 7.7). Our task is to check whether deadlock is possible in such a system. Various steps in using matrix equations are as follows:

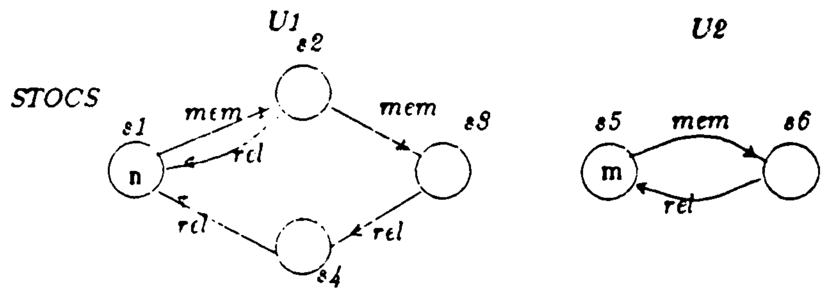


Figure 7.7: Preemptive Memory Server

(1) Let the initial configuration of STOCS for the problem be M_i which is $(n,0,0.0,m,0)$ for our example. We are interested in knowing if there is a possibility of deadlock. It is easy to check that the only possible deadlock states are: (n,0,0,0,0,m) and (0.0,y,n-y,m,0). Thus the question of deadlock reduces to the question of reachability of any of the above states for some value of n, m and y.

(2) We assign a variable X_i for each possible occurrence of a handshake. This variable represents the number of times that particular handshake must occur to reach the final configuration. For example, X_1 represents the number of times transition *mem* occurs from state s_1 and X_2, the number of times from state s_2. Similarly, *rel* occurs X_3 times from state s_2, X_4 times from state s_3, and $X5$ times from state s_4. Due to the synchronous nature of execution, we conclude that the transition *mem* from state s_4 must have occurred X_1+X_2 times and transition *rel*, X_3+X_4+$X5$ times.

(3) Each of the above handshakes has an additive effect on the configuration. For example, the handshake *mem* from state s_1 has the effect of removing one token from state s_1 and adding one to s_2 (represented as (-

1,1,0,0,0,0)). If the final state is reachable then the cumulative effect of all handshakes can be written as:

$$
\begin{aligned}
M_f = M_i &+ X_1(-1,+1,0,0,0,0) \\
&+ X_2(0,-1,+1,0,0,0) \\
&+ X_3(+1,-1,0,0,0,0) \\
&+ X_4(0,0,-1,+1,0,0) \\
&+ X5(+1,0,0,-1,0,0) \\
&+ (X_1+X_2)(0,0,0,0,-1,+1) \\
&+ (X_3+X_4+X5)(0,0,0,0,+1,-1)
\end{aligned}
$$

This can also be written as $\Delta M = AX$ where $\Delta M = M_f - M_i$, A is the appropriate matrix calculated from above and $X = (X_1, X_2, X_3, X_4, X5)$.

For our example. A is as follows: $A = \begin{bmatrix} -1 & 0 & 1 & 0 & 1 \\ 1 & -1 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ -1 & -1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 & -1 \end{bmatrix}$

Note that A is independent of symbols used and therefore can be calculated just from the structure of the STOCS machine.

(4) We next check whether the above system of equations has a non-negative integral solution. For our example, we find that a solution is not possible unless n=0. Now it is easy to show that if the system of equations does not have a non-negative integral solution then the final configuration M_f is not reachable. Hence, we deduce that the configurations (n,0,0,0,0,m) or (0,0,y,n-y,m,0) are not reachable and therefore the system is free from deadlock.

The chief disadvantage of this method is that even though a non-negative integral solution to above equations may exist, the final configuration may not be reachable. This problem is the same as the one faced during the analysis of Petri nets using Matrix Equations[Murata 84].

3.2. Broadcast Topology

This topology assumes a synchronous broadcast primitive which is an extension of one-to-one synchronous i/o of CSP. The usefulness of such multiprocess synchronization constructs is discussed in [Ramesh 86]. A synchronous broadcast requires that a message be sent only if all other processes are ready to receive it. We will use the example of readers writers problem [Courtois 71] to show that symbolic reachability can be useful for algorithms that use broadcast topology.

The readers writers problem is as follows. Assume that a database is being updated by certain processes called writers and consulted by some other processes called readers. To avoid any concurrency conflict, these processes can access the database only with the following constraints:

(1) At most one writer can access the database. (w-w conflict)

(2) A reader and a writer cannot access the database at the same time. (r-w conflict)

(3) Multiple readers are allowed to access the database at the same time. (r-r okay)

(4) To avoid the possibility of starvation of a writer, we add the constraint that a reader cannot enter the critical region if a writer is waiting to

enter it.

For the analysis of broadcast topology, we will constrain a unit of a STOCS machine to have at most one token. For the readers/writers problem each process forms a separate unit as it interacts with every other process. The STOCS machine for a single reader and a single writer is shown in Figure 7.8. τ events used in the Figure 7.8, are internal to the process and do not need any interaction with the external world. A minus sign before a symbol (e.g. *-enter*) indicates that it is a broadcast and can take place only if other processes can take the corresponding plus transition (*+enter*). We define the ***status*** of a network of processes as a tuple with the number of processes that are in the i^{th} state† as its i^{th} coordinate. Thus, (w,0,0) means that w processes are in state 1, and there are no processes in state 2 and 3. The status reachability of the above problem is shown in Figure 7.8. The analysis shows that (1) the system does not have any deadlock state (2) there is at most one writer in state s_2 at any time, and (3) readers and writers are never in state s_2 at the same time. Note that the reachability graph has a special edge for letting i readers read the database at the same time. Such an edge is added if a configuration C'_2=((w,0),($r-1$,1,0,0)) is reachable from some configuration C'_1=((w,0),(r,0,0,0)) such that there is a decrease in the value of only symbolic coordinates of C_1 (i.e. r). Such an edge corresponds to i instances of a transition.

† a process is said to be in state i if its token is in state i.

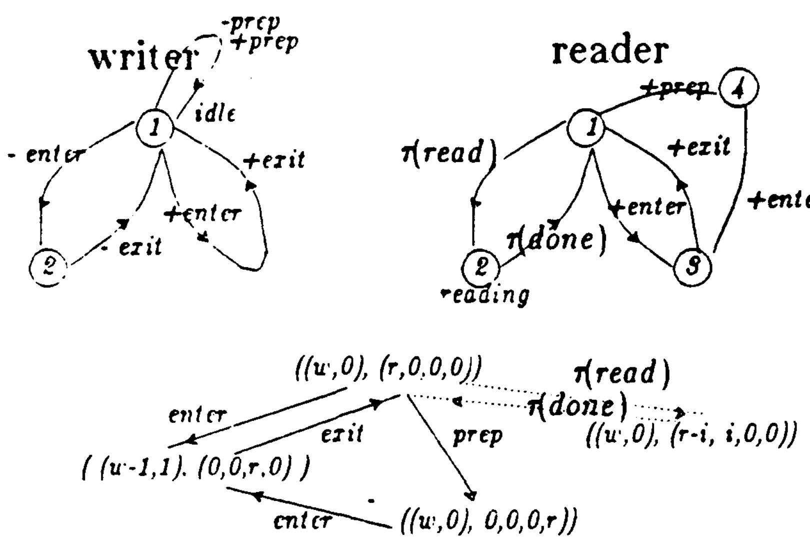


Figure 7.8: Readers Writers Problem

3.3. Ring Topology: Induction Analysis

As the number of processes in a network (say, n) may be a large, variable or unknown quantity, the construction of the global state graph is not feasible. It is desirable to have a method that analyzes the network with a small number of processes and then generalizes results to a larger n. The key idea that can be frequently applied for many systems is that of *induction*. Instead of studying the system with a large or an unknown number of processes, this method analyzes it with a small number of processes and then the invariance of assertions is analyzed with the increase in the number of processes. Since the analysis is done for a small number of processes, the reduction in the global state space is substantial. The principle of induction states that if an observer cannot distinguish between two systems with i and $i+1$ processes connected in a linear fashion even with an infinite input to both systems, then he also cannot distinguish between a system with i processes and any other system with more than i processes. It follows that it is sufficient to analyze the network with $i+1$ processes for any input-output assertion on more than i processes. This principle is illustrated in Figure 7.8.

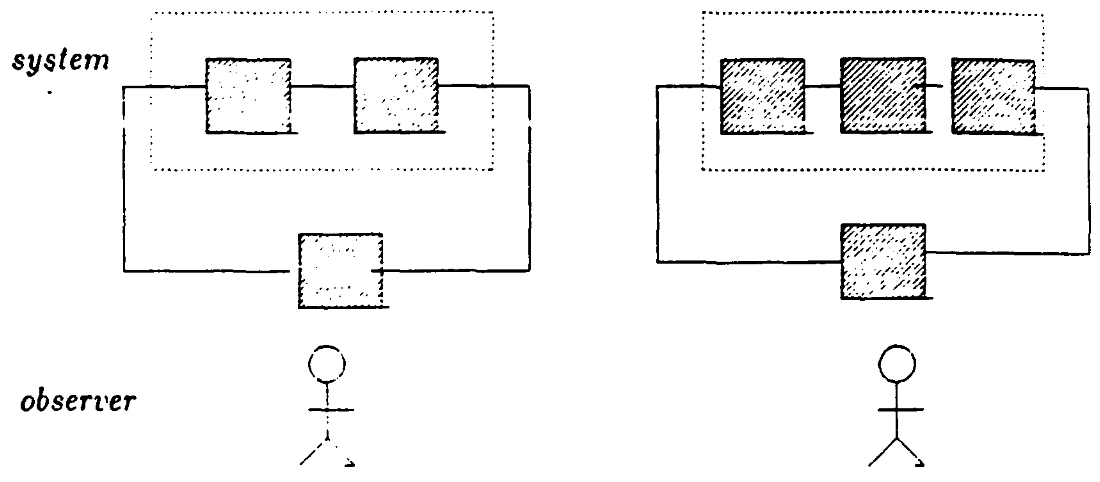


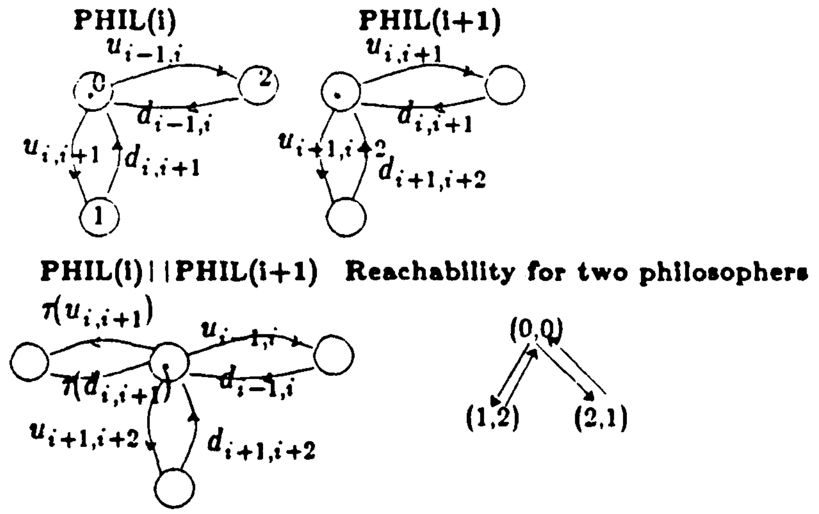
Figure 7.8: Induction Principle for Ring Topology

We illustrate this by analyzing the dining philosopher's problem, of which, Hoare [Hoare 85] remarks that, "*There is no hope that a computer will ever be able to explore all these possibilities (for a deadlock). Proof of the absence of deadlock, even for quite simple finite processes, will remain the responsibility of the designer of concurrent systems.*" To arrive at this conclusion, he computes the number of reachable states of philosophers by taking the product of their states. We claim that with a smarter way of enumerating possibilities, a computer need not explore all of them. For example, we could analyze our algorithm for two dining philosophers instead of five philosophers which will bring the number of possible states to a small quantity. We, of course, have to show that the analysis does not change with the number. We next describe the problem, a deadlock free solution and its automatic analysis.

3.3.1. Simple Induction

This problem, due to Dijkstra, requires an algorithm for philosophers who are sitting around a circular table. There are five philosophers and five forks, each of which is between two philosophers. There is a bowl of spaghetti in the center which can be eaten by any philosopher but its tangled nature requires that a philosopher use both his left and right forks.

A solution which assumes synchronous communication is as follows. A philosopher, when hungry, either picks up both the forks simultaneously or waits for them to be available. This way of picking forks guarantees that there will not be any deadlock. To express our solution, we assume that i^{th} philosopher *owns* i^{th} fork and needs to ask only the right neighbor for the use of $i+1^{th}$ fork. For convenience we will use $u_{i,j}$ to denote that i.picks up fork.j and $d_{i,j}$ to denote that i.puts down fork.j. With this notation, Figure 7.9 shows the solution expressed in the STOCS model.



Notation
$u_{i,j}$ = i.picks up fork.j
$d_{i,j}$ = i.puts down fork.j

Figure 7.9: Dining Philosophers: Analysis

To show that the solution is deadlock free, we could use a computer to explore the reachable states. In the past, automatic analysis meant exploring the cross product of all possible states of five philosophers and five forks (or hundred philosophers and hundred forks for a hundred philosopher problem). Our technique, in contrast, exploits the symmetry in the problem so that the complexity of analysis for five philosophers is the same as that of, say, one hundred philosophers. Various steps in our technique are as follows:

(1) Let $SYS_k=(PHIL_i||..||PHIL_{i+k-1})$. Find the smallest value of k for which $SYS_k=SYS_{k+1}$. For most symmetric cases $k=1$ or 2 will suffice. For dining philosophers, $SYS_1=SYS_2$ as shown in Figure 7.10.

(2) To analyze a ring with any number of units, say n, it is sufficient to analyze it with $k+1$ units. Thus, for our case it is sufficient to analyze the system with two philosophers to make any assertion about a system with five or one hundred philosophers.

(3) We next construct a reachability graph for two philosophers and find that there is no state with out-degree equal to zero (see Figure 7.10). We conclude from this that the system with five philosophers will also be deadlock free.

3.3.2. Induction Analysis with Filters

Observe that simple induction required that the observer not be able to detect the difference on *any* input. This constraint may prove too restrictive to apply induction techniques for certain problems. Therefore, we relax the condition using the concept of *filters*. Filters are formal mechanisms to capture the condition that not all inputs may be possible for the system and therefore we are willing to call two systems equivalent as long as their outputs do not differ on possible inputs.

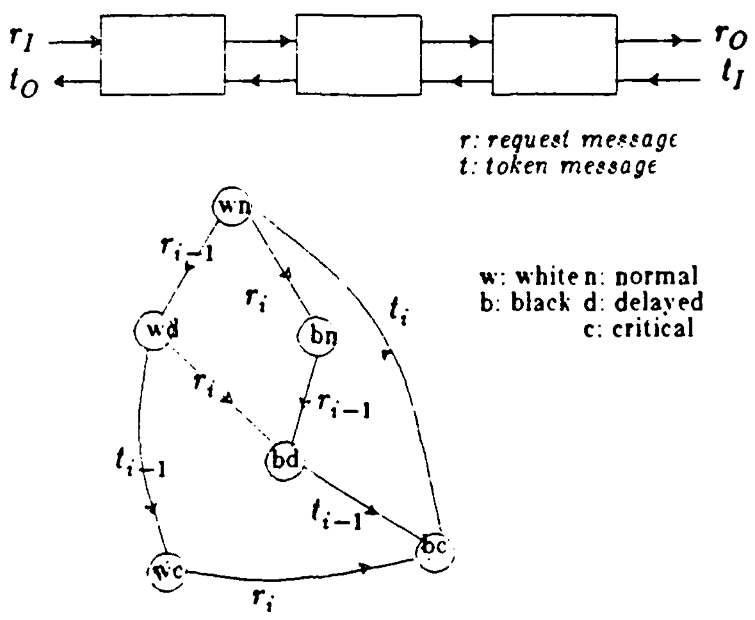
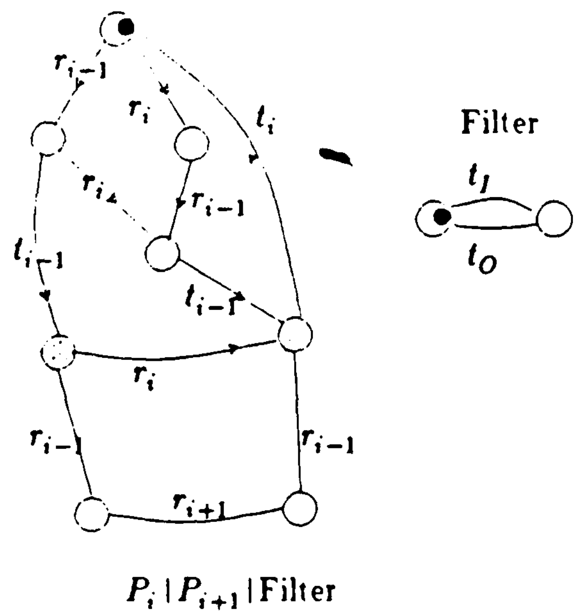


Figure 7.11: Mutual Exclusion in a Ring

We illustrate the use of filters by a mutual exclusion algorithm in a ring network. Clarke et.al.[Clarke 86] use the same example to illustrate their manual induction technique. Dijkstra[Dijkstra 86] also uses the same example to show how regular expressions can be used to prove the correctness of certain algorithms. His proof, again, is manual. The mutual exclusion problem in a ring of processes is as follows. The machines are connected in a ring fashion and can communicate with their neighbors. Each process can be in one of the three states: normal (n), delayed (d) or critical (c). A process can execute the critical region only if it is in the critical state. The objective is to ensure that at any time at most one machine is in the critical state. We introduce the notion of a token which is held by a single machine. To avoid passing tokens unnecessarily, we introduce a request signal which indicates an interest in the token. A process that wants to execute the critical region and does not have the token gets delayed. Following Dijkstra's algorithm, tokens are sent to the left, whereas request signals are sent to the right (see Figure 7.11). We color each of the process as white or black depending upon whether an interest in the token exists to the left. Figure 7.11 shows the example of a distributed mutual exclusion algorithm in a ring network expressed in the STOCS model.

If we try to apply the induction technique that was used for dining philosophers we find that step 1 is not applicable, that is, there does not exist any k for which SYS_k is the same as SYS_{k+1}. This can also be seen intuitively from the algorithm. An observer can detect the number of processes he is connected to by sending multiple token messages. The number of processes in a system would be equal to the maximum number of token messages that are absorbed by the system.



$P_i \mid P_{i+1} \mid$ Filter

Figure 7.12: Composition of two processes with the filter

To solve this problem, we use the notion of filters to constrain the observer to send at most one more token message than he receives from the output. We now show the steps in the modified induction technique using the mutual ring example.

(1) Model all the constraints on the input output behavior through a process called FILTER. Figure 7.12 shows such a filter for our example.

(2) Verify that a process in the system indeed satisfies the constraint imposed by the filter. If we substitute all request messages in an ENTITY by ϵ, t_{i-1} by t_I and t_i by t_O we do get the filter as a result.

(3) Find the smallest k such that a filtered system with k units is identical to a filtered system with $k+1$ units. That is,

$FILTSYS_k=(ENTITY_i||....||ENTITY_{i+k-1}||FILTER)$.

For our example we find that $FILTSYS_1 \neq FILTSYS_2$ but $FILTSYS_2=FILTSYS_3$. It is easy to check that $SYS_k \neq SYS_{k+1}$ for any value of k.

(4) Thus from the principle of induction we deduce that it is sufficient to analyze the algorithm with three processes to make an input-output assertion on any number of processes greater than three.

4. Conclusions

Due to decreasing costs of hardware and advances in VLSI technology, systems with multiple processes have become popular. Typically, a concurrent algorithm on such architectures consists of multiple processes each of them executing a very simple procedure. Examples of such paradigms of computation are Hypercube algorithms and Connection Machine algorithms. There is an acute need for systems that can analyze such distributed systems. Automatic analysis of even finite state systems runs into the problem of state space explosion. Since most distributed systems show symmetry, we suggest techniques that exploit symmetry to reduce the state space. STOCS is a useful model to represent symmetric distributed systems. We use symbolic reachability and matrix equations to analyze systems expressed in the STOCS model with star or broadcast topology. We use induction to reduce the number of process that need to be analyzed in a ring network.

CHAPTER 8

ConC: Embedding of STOCS in C

1. Introduction

The availability of cheap hardware and communication facilities has made distributed systems an attractive proposition. However, the difficulty of concurrent programming has kept it away from average programmers [Chandy 85]. In present concurrent programming languages systems, the communication aspects of a program are interwoven with computational aspects. As a result, any analysis of the communication structure of the program is difficult. By analysis, we mean questions such as - "Is a certain sequence of events possible?", "Is a certain state reachable?" etc. To make concurrent programming easier, the system should provide automatic analysis of the communication aspects of a program. In addition, the communication aspects of the software should be specified at a very high level of abstraction. Therefore, we had two goals in designing the communication primitives - *high level specification* and *analyzability*.

In this chapter, we propose two new constructs for concurrent programming - *handshake*, a generalization of the remote procedure call, and *unit*, a communication structuring mechanism. A handshake is shared among two or more processes. Each process has a procedure-like interface with a handshake. When all the participating processes call their handshake procedures, the shared handshake body is executed. The unit construct is used to restrict the sequence of possible calls to various handshake procedures and thereby provide a synchronization mechanism between multiple processes. Thus, a unit can be viewed as an automaton that specifies all possible sequences of handshake procedures. The handshake and unit constructs form part of the STOCS Model. Since STOCS machines are theoretically equivalent to Petri Nets, all the analysis techniques for Petri nets such as coverability tree [Karp 68] and matrix equations [Murata 84] are directly applicable to the STOCS.

In our paradigm, we support separation of concerns by separating *internal* objects and *external* objects. Internal objects are specified in any standard sequential programming language such as Pascal, C or sequential Ada. These objects are used mainly to capture the computation aspects of the system and do not concern themselves with either synchronization or communication. External objects, on the other hand, are written as units and handshakes. They specify the computation that is directly related to communication. For example, synchronization is handled by these objects. They are mechanically analyzable for most interesting properties as their expressive power is less than that of Turing machines,

The rest of the chapter is organized as follows. Section 2 discusses the related work in the area of constructs for concurrent programming. In section 3, we discuss our constructs for concurrent programming. Section 4 discusses the interaction between computation and communication objects in our paradigm. Section 5 discusses the status of the ConC project.

2. Related Work

Andrews and Schneider [Andrews 83] classify concurrent languages into three categories. The ***shared memory based*** programming languages assume that variables can be accessed by any process. To guarantee mutual exclusion, constructs such as critical regions and monitors are used. Example of such languages are Concurrent Pascal, Mesa [Mitchell 79] and Modula. *Message based* programming languages provide send and receive constructs for communication. Examples of such languages are CSP [Hoare 85] and PLITS [Feldman 79]. *Operation based* languages combine aspects of the other two classes. They provide remote procedure call as the primary means of process interaction. Ada, Distributed Processes [Brinch Hansen 78] and SR [Andrews 82] fall in this class. Since the handshake construct extends the remote procedure call for multi-party interaction, it belongs to this class as well. The features that distinguishes ConC from related efforts are as follows:

(1) **Synchronous Communication**: We believe that programmers of distributed systems should not have to deal with asynchronous communication as it makes a program difficult to debug. and analyze. In this respect, we differ from PLITS, and agree with the philosophy of programming languages such as Ada and CSP.

(2) **Multi-Process Interaction**: Many applications require interaction between more than two processes and the user can program at a high level if such a facility is directly provided by the language. CIRCAL [Milne 85], Raddle [Forman 86], Multi-way Rendezvous [Charlesworth 88], PPSA [Ramesh 87], and Script [Francez 83] have also suggested multi-party interaction in one form or another. CIRCAL, Raddle and PPSA allow synchronization based on matching of event names but do not provide a remote procedure call-like

interface. Script shows how details of multi-process interaction can be hidden but does not provide direct support for the multi-party interaction. None of them supports any form of analysis.

(3) **Analysis of Interaction**: As most errors in concurrent systems arise due to erroneous specification of process interaction, any analysis of the interaction will greatly increase the programmer's productivity. None of the above mentioned languages supports analysis. Such analysis is more common for communication protocols which is done mainly for specifications expressed in State Machines, Petri nets or bounded variable programming languages [Sunshine 78]. One of the early attempts to incorporate such analysis in a full fledged programming language was Path Expressions [Campbell 74]]. Path Pascal [Campbell 79] based on Path expression is, however, a shared memory based language. Path expressions are also cumbersome to write and understand for even slightly complex constraints. In addition, the analysis provided by Path Pascal is not as extensive as that provided by ConC.

(4) **Communication Abstraction Mechanism**: Researchers in programming languages have found abstractions a useful mechanism to increase the understandability of the software. Consequently, current programming languages provide control abstraction through loop constructs and procedure calls, and data abstraction through abstract data types. One of the main functions of an abstraction is to provide only structured access to the primitives. For example' a control abstraction mechanism seeks to provide a structured use of goto's. Similarly, the complexity of concurrent software has made it necessary that goto's of the communication world (send, receive, remote

procedure calls etc.) be allowed only in a structured manner. Path expressions specify the sequence of procedures that can be made on shared variables and therefore can be termed as the first attempt for providing such a mechanism. Francez and Hailpern [Francez 83] were the first to coin the term and use it in their proposal of Script. ConC provides structuring of the communication primitives through the unit construct.

Table 1 summarizes some of the well known concepts that can be shown to be special cases of constructs provided in the ConC.

Feature	**Example**	**ConC**
Synchronous communication	CSP	handshake
Remote procedure call	Ada	parametrized handshake
Multi-process interaction	Raddle	multi-process handshake
Abstraction Mechanism	Script	unit
Path Constraints	Path Pascal	unit expressions
Reachability	Petri Nets	STOCS

Table 8.1: Special Cases of Handshake and Unit Constructs

3. Constructs

3.1. Handshake Construct

The remote procedure call has become one of the most favored communication primitive because of its similarity to the local procedure call, a well understood concept. A handshake is a remote procedure call generalized for multiple parties.

A handshake consists of the declaration of handshake procedures and a shared body. The body of the handshake is executed only when all handshake procedures have been called by their respective processes. Thus, handshake can be used as a synchronization point of multiple proceses. For illustration, consider the distributed players problem. Assume that there are four players who are interested in playing various games as shown in Figure 8.1. Joe is willing to play chess, bridge or poker. Mary is willing to play any of the games while Jack and Bob play only bridge or poker. Playing a game requires *rendezvous* between two or more processes. This is achieved by handshake construct as shown in Figure 8.2.

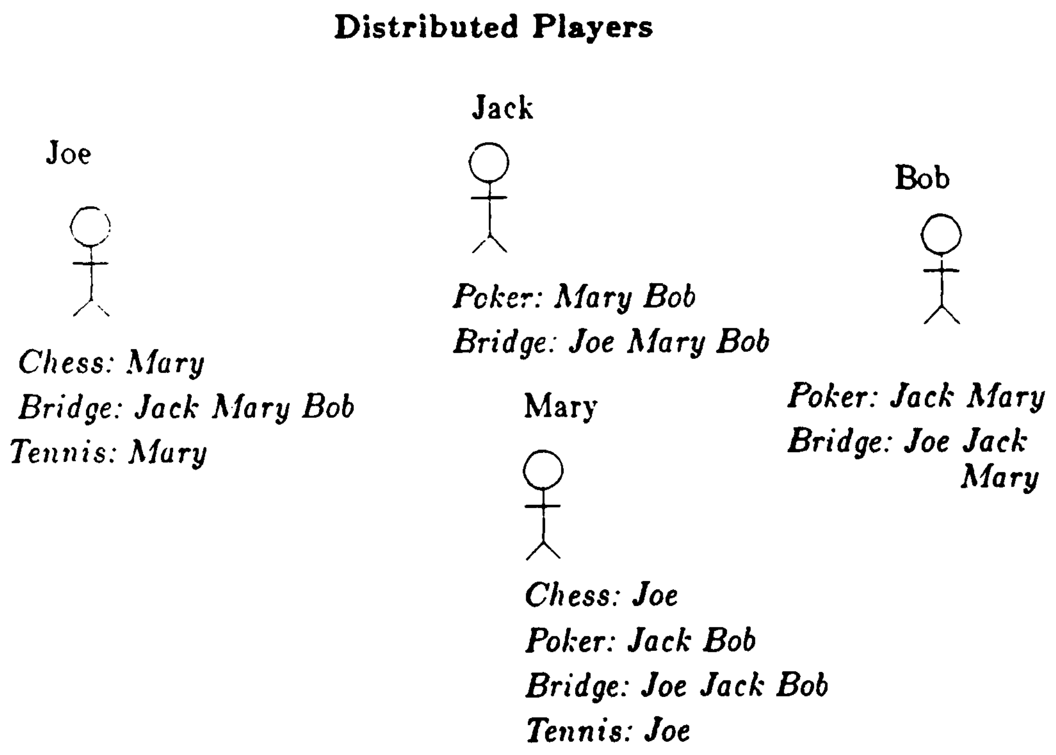
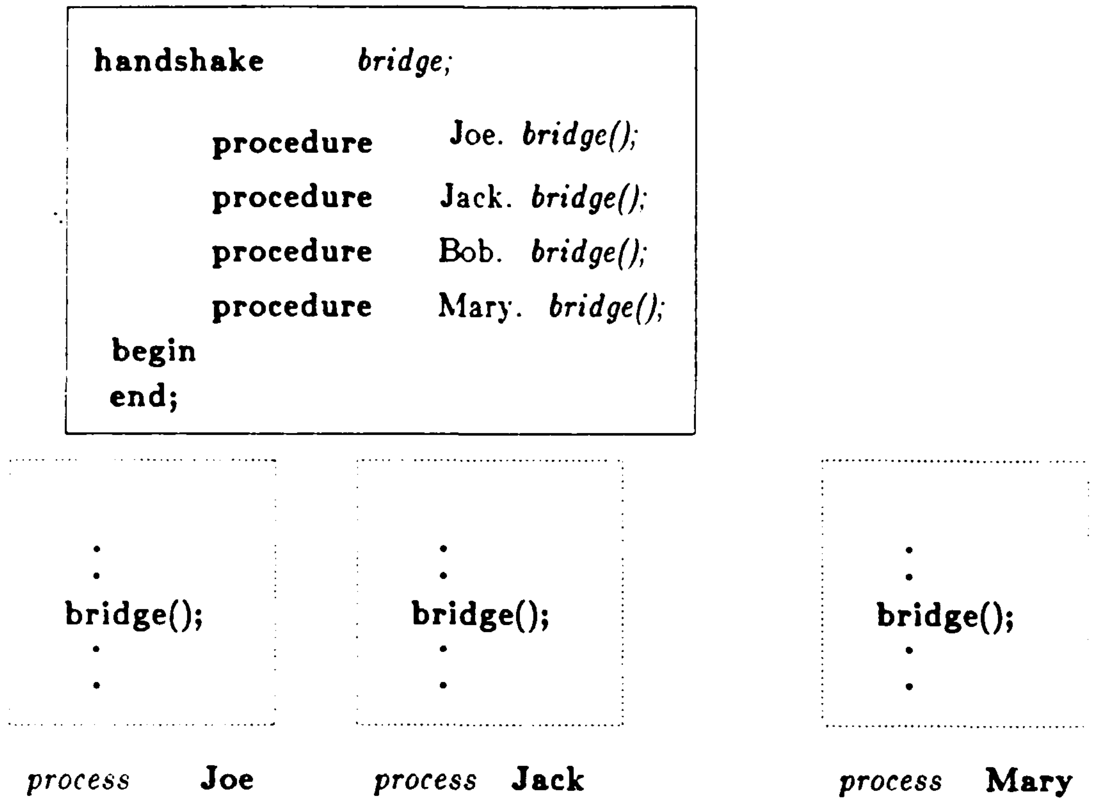


Figure 8.1: Distributed Player Problem

The above example illustrated the use of the handshake construct for synchronization. The handshake construct is also useful for communicating data from one process to the other. The handshake procedures may be called with parameters. When the handshake is executed by the master of the handshake,



```
handshake      bridge;

        procedure    Joe. bridge();
        procedure    Jack. bridge();
        procedure    Bob. bridge();
        procedure    Mary. bridge();
 begin
 end;
```

Figure 8.2: Handshake Construct for Distributed Player Problem

all the parameters are considered available. The body of the handshake can use any of the parameters or its own local variable. As an example of a handshake with parameters, consider the same example of distributed players. Assume that Joe decides where they should meet for the game of bridge. The revised handshake declaration is shown in Figure 8.3.

```
handshake bridge;
procedure Joe.bridge(Joeplace: alpha);
procedure Jack.bridge(var Jackplace: alpha);
procedure Mary.bridge(var Maryplace: alpha);
procedure Bob.bridge(var Bobplace: alpha);
begin
  Jackplace = Joeplace;
  Maryplace = Joeplace;
  Bobplace = Joeplace;
end;
```

Figure 8.3: Handshake with Parameters

We next describe the syntax for the handshake construct using BNF. We use {} to denote zero or more repetitions of the enclosed expression. Note that the syntax of a handshake is symmetric for caller and callee in contrast to Ada's rendezvous where the callee uses accept and the caller uses entry procedure call to make a rendezvous.

```
; a handshake specification
<handshake-dcl> ::= handshake id ';' <global-declaration>
    {<proc_specs>} <local-declaration> <body> ';'

; this section specifies types used in declaring parameters
<global-declaration> ::= the usual const and type declarations

; headers for various procedures which share the body
<proc_specs> ::= procedure id ( { <param> } ) ';'
<param> ::= [var] id ':' <type> ';'

<local-declaration> ::= local variable declaration

; the body is executed when the handshake takes place
<body> := the usual programming language body
```

As another example of the handshake construct consider the synchronous *send* provided in the Unix as a library facility. The handshake description of such a primitive in ConC is shown in Figure 8.4. It specifies that when process P1 calls *send* and P2 calls *receive* with parameters, the associated body with the handshake is executed by the first process named in the handshake (P1). Figure 8.5 shows handshakes when there is a buffer process that can store messages. For simplicity, we assume that the processes are interested in communicating integers only. These examples illustrate that handshake construct can simulate messages easily.

We impose certain restrictions on use of the handshake construct. The syntax requires every participant in the process to be explicitly named. Similarly, a handshake procedure cannot be called from within the body of another handshake. These restrictions are required for the feasibility of automatic analysis of the communication structure.

```
handshake syncsend;
const
  MAXLENG = 50;
type
  message = array[1..MAXLENG] of char;
  numbytes = 0..MAXLENG;

procedure P1.send( senddata: message; scount: numbytes);
procedure P2.receive( var recdata: message;
                      var rcount: numbytes);

var i: integer;
begin
  for i:=1 to scount do
    recdata[i] := senddata[i];
  rcount := scount;
 end;
```

Figure 8.4: Synchronous Send

```
handshake put_item;
  procedure sender.send(sdata: integer);
  procedure buffer.insert(var bdata: integer);
  begin
    bdata := sdata;
  end;

handshake get_item;
  procedure buffer.remove(bdata: integer);
  procedure receiver.receive(var rdata: integer);
  begin
    rdata := bdata;
  end;
```

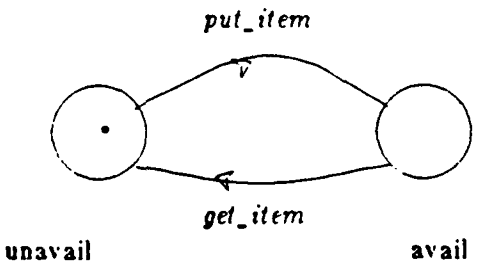
Figure 8.5: Asynchronous Send

Unit Specification

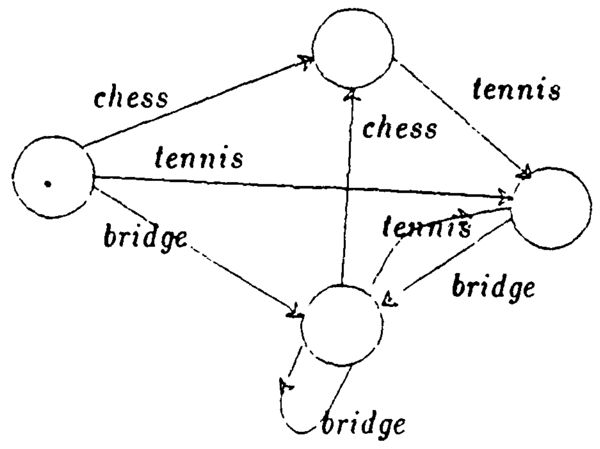
In the example of distributed players, players may have different constraints on their sequence of games. For example, Joe may wish to play only tennis after chess. Similarly, in the example of buffered send, we did not specify the buffer process. If the buffer process allowed *put_item* and *get_item* in any order, the communication may be faulty. A single-buffer process behaves correctly if it satisfies the constraint that a *put_item* is always followed by a *get_item* and *vice-versa*. As a result, the sender may have to wait for the receiver to read an item from the buffer process before it sends another item. To express such constraints and therefore provide a high level synchronization mechanism, we provide the *unit construct*.

To describe all possible sequences of handshake procedures, we can use a algebra based model (e.g. regular expressions) or transition based model (e.g. finite state machines). The transition based model has the advantage that it is is graphical, while the algebra based model is sometimes more natural to the application. Our implementation uses unit machines, the transition based model, for expressing the constraints. We can also use unit expressions because a unit machine can be converted to a unit expression and vice-versa.

A unit is a directed graph where vertices are called places, and edges between them are labeled by names of handshakes. In addition, there is the concept of tokens which may be thought of as residing in places. A handshake can take place only if there is a token in the tail vertex (source place) of the handshake. After execution, the token moves to the head vertex (destination place). Figure 8.6 shows the linguistic and graphical equivalent of the constraints imposed by Joe. Figure 8.7 shows the linguistic and graphical equivalent of a one-frame buffer. The marking construct is used to describe the number of tokens at various places. The body of a unit consists of enumeration of all transitions in the unit. These transitions are arranged on the basis of their source places. A place name is followed by the description of transitions, each consisting of a handshake name followed by the destination place.



- *Joe will play only tennis after chess*
- *Joe will play only bridge after tennis*



```
unit Joecomm;
  marking [start:1];
  begin
  start
    > chess cstate;
    > tennis tstate;
    > bridge bstate;
  cstate
    > tennis tstate;
  tstate
    > bridge bstate;
  bstate
    > tennis tstate;
    > chess cstate;
    > bridge bstate;
end;
```

Figure 8.6: Unit Specification for Joe

```
(* put_item should be followed by a get_item
     get_item should be followed by a put_item *)
unit buffercomm;
 marking [unavail:1];
 begin
  unavail
    > put_item avail;
  avail
    > get_item avail;
 end;
```

Figure 8.7: Unit Specification of a One-frame buffer

Figure 8.8 presents the use of *-places to specify the unbounded buffer problem in ConC.

```
(* the receiver must wait for the sender *)
unit buffercomm;
 marking [unavail:*];
 begin
  unavail
    > put_item avail;
  avail
    > get_item unavail;
 end;
```

Figure 8.8: An Example of the Unit Specification

The BNF for the specification of a unit is as follows:

```
<unit_specs> ::= unit id ';' <marking>
             { <transitions> } end ';'
<marking> ::= marking
          { '[' <placename> ':' <num> ']' } ';'
<num> ::= '*' | integer
<transitions> ::= placename
              { '>' transname placename ';' }
```

3.2. Guard Construct

For selective communication, we also assume that the language has the guarded command construct as proposed by Hoare for CSP. A guarded command consists of one or more <guard, statement> pairs. A guard consists of a boolean condition and optionally a handshake. The handshake is enabled only if the boolean condition is true. If an enabled handshake can be executed (participating processes are willing to execute the handshake), the guard is considered true and the statement corresponding to the guard can be executed.

The syntax of the guard construct is as follows:

```
<guarded_command>:: '[' <guard> '->' <statement> ']'
<guard>:: <boolean_condition> '&' handshakeid
```

For an example of guard construct, consider the buffer process which may communicate with either the sender or the receiver. Its specification is as follows:

```
int findex = 0;
int bindex = 1;
[
put_item -> insert(item);
          findex = (findex + 1) mod size;
          buffarray[findex] = item;
get_item -> remove(buffarray[bindex]);
          bindex = (bindex + 1) mod size;
]
```

3.3. Mutual Exclusion between Two Processes

As an example of these constructs, consider the mutual exclusion between two processes X and Y. The entire system has four handshakes - p1in, p1out, p2in, p2out. P1in handshake requires participation from both the processes X and Y. This is specified in the handshake declaration of p1in. P1out, on the other hand, does not need any coordination from the process Y. The unit construct allows p2in to happen only if the process X is in a non-critical state. The entire specification of the process X is given in Figure 8.9.

```
handshake p1in;
   procedure X.p1in();
   procedure Y.p1in();
  begin
  end;

handshake p1out;
   procedure X.p1out();
  begin
  end;

(* communication unit for process X *)
unit mutex1;
  marking[noncritical:1];
  noncritical ;
       > p1in critical;
       > p2in noncritical;
  critical ;
       > p1out noncritical ;
end;

(* internal computation for process X *)
main()
{
 int i;
 for (i=1; i<=10; i++)
 {
   p1in();
   (* this is the critical region *)
   p1out();
}
```

Figure 8.9: Mutual Exclusion Between Two Processes

3.4. Dining Philosophers

A deadlock free solution to the dining philosopher problem (discussed in Section 7.3.3) expressed in ConC is shown in Figure 8.10. $get_{i,i+1}$ represents that i^{th} philosopher has taken possession of $i+1^{th}$ fork. The $philosopher_i$ does not seek possession of $i+1^{th}$ fork unless it also possesses i^{th} fork. Note the simplicity of the solution due to the availability of synchronous communication.

```
handshake $get_{i,i+1}$
  procedure $philosopher_i.get_{i,i+1}$;
  procedure $philosopher_{i+1}.get_{i,i+1}$;
 begin
 end;

unit $philunit_i$;
      marking[neutral:1];
      neutral
            > $get_{i,i+1}$ eating ;
            > $get_{i-1,i}$ waiting;
      eating
            > $put_{i,i+1}$ neutral;
      waiting
            > $put_{i-1,i}$ neutral;
end ;

process $philosopher_i$;
begin
   if hungry then begin
      $get_{i,i+1}$();
      eat();
      $put_{i,i+1}$();
   end;
end;
```

Figure 8.10: A Solution To Dining Philosophers Problem

Having stated a solution to the dining philosophers problem, we would like to verify that our solution is indeed deadlock free. Current programming systems typically require manual analysis for such questions. As we stated earlier, one of our aim is to automatically analyze specifications expressed in handshake and unit constructs. We can do so because these constructs are based on the STOCS Model.

4. Interaction between Computation and Communication Objects

Each logical process is actually composed of two real processes: computation and communication process. The computation process communicates only with its communication process, which in turn communicates with other communication processes. Therefore, the actual communication between various processes is as shown in Figure 8.11.

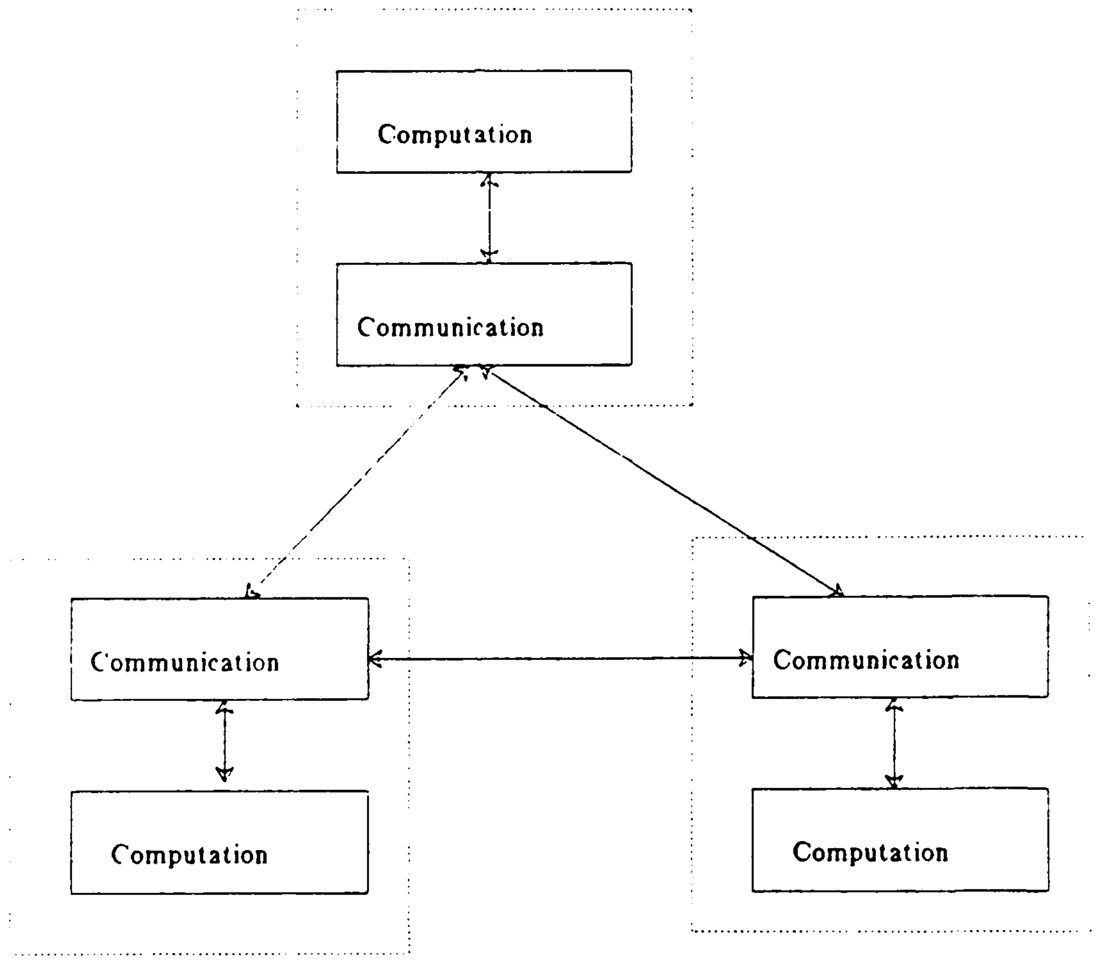


Figure 8.11: Communication Structure of ConC programs

The computation process interacts with communication process by two means:

(1) Simple handshake call: As seen earlier the execution of a handshake may require the participation of multiple processes. The computation process sends

an **enable** message to the communication process whenever it is ready for a particular handshake and waits for a reply from it. The communication process goes through a series of protocol messages with other communication processes to agree on the execution of the handshake. If it succeeds, it tells the computation process to proceed and send the relevant message to the master of the handshake. If the handshake is not possible because one of the particpant process has terminated then the communication process sends an error message to the computation process.

(2) Calls from Guard: Since only one handshake is allowed in every guarded statement, we conclude that if all participant processes are ready for a handshake it can be always be executed, even if the handshake call is from a guard. The computation process enables all the handshakes that are called from the conditions of the guarded statements. It then waits for a reply from the communication process. The communication process sends to the computation process, the name of the handshake it has committed. It is the responsibility of the computation process to execute the handshake.

5. Implementation of the ConC System

The current ConC system consists of two sub-systems: *ConC translator, and STOCS analyzer.* ConC translator generates a set of "C" processes from a ConC program. These processes communicate using the semantics of a synchronous handshake in STOCS. The execution of a handshake requires synchronization between multiple processes similar to that required by a generalized CSP alternative command. Chapter 9 describes an algorithm for multi-

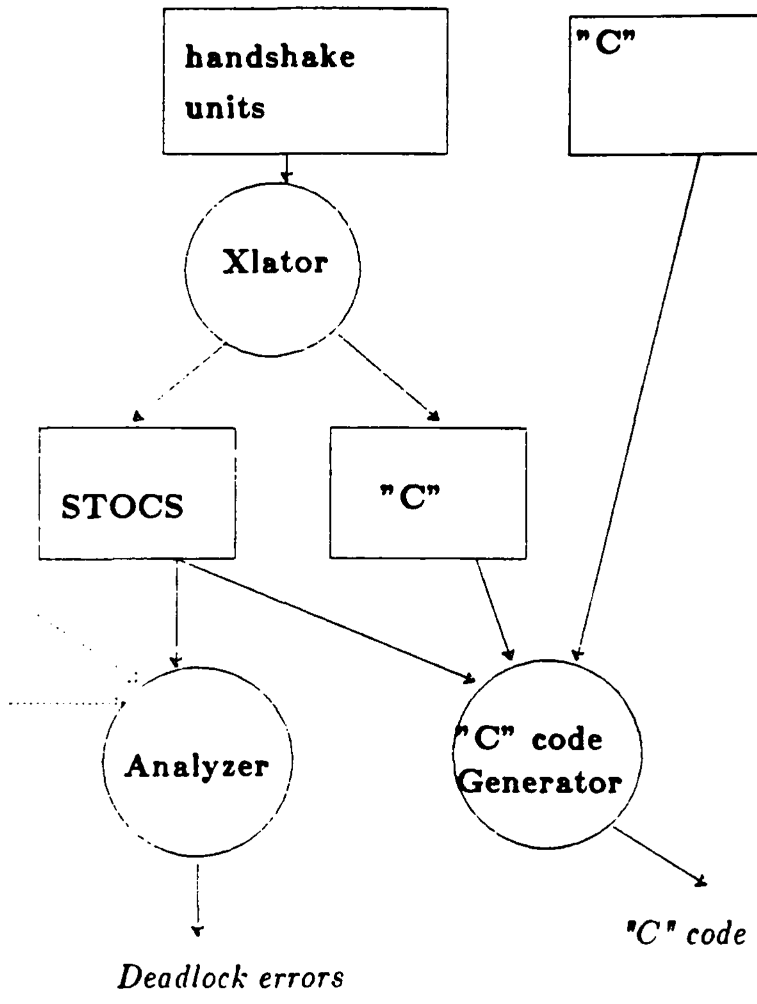


Figure 8.12: The Architecture of ConC System

process synchronous communication. ConC Translator is implemented on SUN workstations with 4.2 BSD UNIX. STOCS analyzer analyzes a given STOCS for the following type of queries: Is configuration C1 reachable? Is there any configuration with no exits?(potential deadlocks) It is written in Franzlisp and runs on 4.3 BSD UNIX.

6. Conclusions

This chapter presents two new constructs to support distributed computation - handshake and unit. The handshake construct is a multi-process generalization of the RPC. The unit construct is used to specify the possible sequences of handshakes and thereby provide a synchronization mechanism between multiple processes. These constructs unify a large number of concepts such as semaphores, monitors, path expressions, input/output, remote procedure calls and communication abstraction. These constructs are based on a formal model called the STOCS model which is mechanically analyzable. The analysis can be done with respect to reachable configurations of a STOCS machine and the language accepted by it.

The proposal for ConC is unique in that it combines aspects from diverse languages such as CSP, Ada, SCRIPT, Path Pascal, Raddle and PPSA. The theory combines aspects from algebraic theory, net theory and formal language theory.

CHAPTER 9

Execution of STOCS Machines

1. Introduction

The STOCS model is based on the concept of handshake, a shared event. Therefore, to execute a STOCS machine, we need a mechanism for implementing shared events. Execution of shared events is required in general by distributed systems which often need tricky synchronization between multiple processes. Example of such shared events are: (1) distributed transactions in databases that require commit by either all or none of the processes (2) atomic broadcasts that require that a message be received by either all or none of the receivers. Shared event is such an useful concept that it is not surprising that it appears in coupled state machines, Petri Nets, CSP and CCS. Ease in specification of concurrent systems using the concept of shared event provides a strong motivation for the search of an efficient algorithm for its execution.

Multi-process shared event execution problem is as follows. Let there be n geographically distributed processes. Each process is either waiting for some event or executing. An event may require cooperation of two or more processes. Each process when idle is willing to embark on any of the event that is enabled in its current state. We assume that processes can communicate with each other asynchronously by means of reliable messages. We have to design an algorithm for executing shared events between these processes.

As an example of this problem in real life, consider the distributed players problem. Assume that there are four players who are interested in playing various games as shown in Figure 9.1. Joe is willing to play chess, bridge or poker. Mary is willing to play any of the games while Jack and Bob play only bridge or poker. Joe in state s_2 will play only tennis and only bridge in state s_3. Similarly, Mary plays only poker/tennis if she is in state s_2 and chess/bridge if in state s_3. Since games require cooperation between two or more players the players may have to wait for each other. Also the players are in different cities (i.e. on different processors) and can communicate only through mail (asynchronous messages).

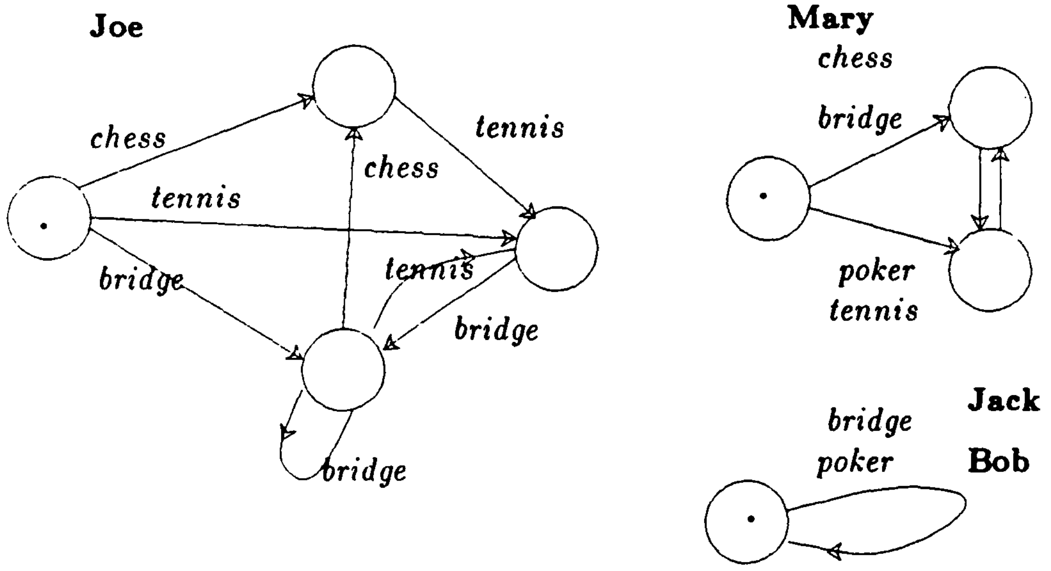


Figure 9.1: Distributed Player Problem

Our algorithm makes following assumption on the communication network:

(1) **Reliable Messages**: The algorithm assumes that all messages sent by one machine to another are received uncorrupted in proper order. A service can easily be provided by a communication protocol layer that detects duplicate, lost, out-of-sequence and corrupt messages.

(2) **Clock Synchronization**: The algorithm assumes that local and global causality as proposed by Lamport is preserved by clocks of various machines. This can easily be provided by an extra layer of clock synchronization that uses Lamport's algorithm.

This chapter is organized as follows. Section 2 describes the related work in execution of shared events. Section 3 presents our algorithm for the execution. Section 4 describes the message complexity of the algorithm. Section 5 proves its correctness. Section 6 explores some efficiency considerations of the algorithm.

2. Related Work

The execution of shared events also arises in implementation of the generalized I/O command of CSP. A CSP program, as described in [Buckley 83], consists of a set of processes that communicate with each other using synchronized message passing. Communication between processes occur when two processes have matching input and output statements. The alternative command of CSP provides non-determinism by letting a process select one of the several statements for processing. Each statement is protected by a guard (a boolean expression and/or one input statement) which must be enabled for the statement to be considered for selection. A guard is enabled if the boolean expression evaluates to true and the named output process has not terminated. However, not all algorithms are easy to express using only the constructs of CSP. Researchers have found it useful to extend the notion of guard to include output command and many implementations have been presented

[Buckley 83, Bagrodia 86, Ramesh 87, Lee 87, Natrajan 86]. This generalized CSP construct is obviously a special case of multi-process synchronous events problem.

[Buckley 83] presented four conditions that should be satisfied by an effective implementation of the CSP I/O construct. They showed that [Silberschatz 79, Snepscheut 81] did not satisfy one or more of these conditions. [Back 84] improved upon this result by providing an implementation that satisfied two more conditions. [Ramesh 87] provided an improved implementation which could be extended to allow multi process synchronization. We provide a new distributed algorithm that has following distinguishing features from rest of the work: Our algorithm satisfies all six conditions, shows strong fairness and is extensible to the case of multi-process synchronization. In addition, it is simpler than algorithms presented in literature[Buckley 83, Silberschatz 79]. We present in this chapter our algorithm, a proof of its correctness, its message and time complexity. The proposed algorithm differs from its predecessor [Ramesh 87] (referred to as R1 in subsequent discussion) which shares the advantage of multiprocess synchronization in the following ways:

(1) **Sequential Capturing**: In R1, a process captures all processes participating in an events sequentially to avoid any deadlock. This can result in a substantial delay for events that is shared by a large number of processes. Since processes are captured for a long time it also means that other processes may have to wait for a long time for captured processes to be released. In our algorithm, a process tries to capture all participating processes in parallel.

(2) **Message Load**: In R1, no specific process is assigned as a master of a handshake. This makes the algorithm look completely distributed at a superficial level by making every process do the work of master. Thus, if a handshake is shared by m processes, each of them may have to do $O(m)$ amount of work for that handshake. We, on the other hand, assign a master for each handshake, thus simplifying the algorithm and reducing the message load. We also provide algorithm for assignment of this coordination such that the maximum load on any single process is minimized.

(3) **Fairness in Execution**: In R1, a guard is chosen at random by each process thereby guaranteeing that any guard has finite probability of being chosen. We, provide a stronger fairness in the sense that we chose a handshake which is coordinated by a master for the longest time. Our definition of fairness implies fairness proposed by R1.

(4) **Timestamped Messages**: R1 does not use timestamps in its algorithm. Our algorithm requires global causality of timestamps and we assume that there is a clock synchronization algorithm such as proposed by Lamport[Lamport 78] running on the network. Since Lamport's algorithm is very simple to implement and does not incur high penalty, this is a not a serious drawback of our algorithm. Besides, due to usefulness of global causality other algorithms may already be using a clock synchronization algorithm. Timestamps are also useful in ignoring outdated message.

(5) **Ready Message**: We use a set of messages called ready messages which increase the probability that a request message succeeds. If a participant

of a handshake sends a ready message to the master, he is marked as ready in a table. For performance reasons, we do not require processes to send "not-ready" messages when they are not ready for a handshake. Thus, an entry in a ready table can be treated only as a hint. If it indicates that a process is not ready for a handshake then this is true for the steady state of the system. However, if it says that a process is ready for some handshake then this must be confirmed by a request message.

3. Description of the Algorithm

Informally, the algorithm is as follows. Each handshake is assigned a master. A master can execute a handshake if all participating processes commit to it. A process can commit to a handshake by sending a *yes* message to its master. A master requests for these messages by means of *request* messages. A *request* message may be either be delayed or responded with a *yes* message, or a *no* message. If all the participating processes commit the master sends a *success* message to them. On receiving a *success* message, a process can execute the handshake.

When a process is in execution state (executing some handshake), it responds to only two kinds of messages- *ready* and *request*. For *ready* message it makes a note in its ready table whereas a *request* message is replied by a *no*.

Once a process comes to an alternative command it first sends *ready* message to all the masters for the guard it is ready to execute. Then if any transition is ready and it sends out *request* messages. After this the process takes an action only on receiving a message.

Some of the features of this algorithm are as follows:

(1) A process can send a *yes* message to at most one master. Thus, a process commits to at most one master. A process that has committed to a handshake can not send *request* message for any other handshake. This way of committing resembles two-phase commit protocol used in databases for implementing transactions. The difference between two problems is that in databases if two transactions t_1 and t_2 are eligible at some state, then the protocol needs to ensure that the final execution can be written either as t_1t_2 or t_2t_1. In our problem, once a t_1 is executed t_2 may not be valid any more. For example, initially both tennis and chess may be eligible, but once tennis is played chess may not be eligible anymore.

(2) A process that has already committed, on receiving a request for another handshake, says no to a younger process and delays the older process. If the process is the master of its committed handshake and the handshake has not received all the yes messages then it is aborted in favor of an older handshake. This way there cannot be any deadlock between different handshakes. This strategy is commonly referred as wait die strategy in databases as discussed in [Eswaran 76].

(3) Processes always include the timestamp of the handshake they are responding to. This has the advantage that the messages that are obsolete can be detected and therefore ignored.

(4) The fairness is based on the principle of serving the master who has served the longer. With each request message for a handshake, the master

sends the time of the event it coordinated before.

The algorithm as shown in Figure 9.2 use the following messages:

ready:

sent by a process to the master of a handshake indicating its willingness to execute the handshake This message can be sent to multiple masters.

request:

sent by the master to processes for the yes/no reply, With a request message, the master also sends the time of the last shared event it executed as a master.

yes: sent by a process to the master of a handshake indicating its willingness to execute the handshake. This message is sent to only one master.

no: sent by a process to the master of a handshake indicating that it has committed for some other handshake

success:

sent by the master to processes asking them to execute the handshake

abort:

sent by the master to processes asking them to abort the handshake

```
Background()
   if (mtype = ready) update(ready_table)
     else if (mtype = request) reply(currmess, no);

Initialize()
   captured = 0; delayed[ ]=0; initialize_guards;
   send ready messages to various masters;
   if any transition is ready then sendrequest(mytrans);

Handle_Ready()
   update(ready_table);
   if (the transition is ready) and
   (I am not exploring any other transition) then sendrequest;

Handle_Request()
     if (guard[trans]=closed) reply(currmess, no);
     else if (mytrans = 0) /* I am not committed */
           mytrans:=trans;
           reply(currmess, yes);
     else if (timestamp[trans] > timestamp[captured])
           reply(currmess, no);
     else if (master[captured] = myid)
             sendabort(mytrans);
           mytrans = trans;
           reply(currmess, yes);
     else delayed[currmess.src]=trans;

Handle_Abort()
   try_another_transition;

Handle_Succ()
   if (captured = currmess.trans)
      taketrans( currmess.trans);

Handle_Yes()
   checklist[currmess.src]=0;
   if all_have_responded_yes
           taketrans(currmess.trans);
           sendsucc(currmess.trans);

Handle_No()
    rstatus[trans][src] = false;
    sendabort(trans);
    try_another_transition;

try_another_transition()
     if any process delayed respond to it;
     else if any transition ready send request
     else send ready to masters which have been sent no
```

Figure 9.2: Algorithm for Execution of Multi-process Events

Some of the data structures are as follows:

ready_table: a table maintained by the master of a handshake. As explained earlier, this table is only one way correct.

delayed_list: list of all masters that have been delayed by me.

guard[handshake]: Is the handshake enabled in my current state.

captured: master that has captured me

4. Message Complexity

The Worst Case

We will calculate the number of messages a process has to handle in the worst case before it is guaranteed to succeed. We first calculate the number of times a handshake can abort. By our fairness rule, a process can be aborted in favor of some other process at most once. Thus, if there are p processes which are master of some handshake, then a handshake of the process must succeed after $p-1$ or less number of attempts. Hence, the number of requests to a process for a guard is less than or equal to $p-1$ and correspondingly the number of aborts is less than or equal to $p-2$. There is at most one success message. Therefore, the number of messages in the worst case for a master guard with d slaves to succeed in executing a handshake is:

ready messages at most $(p-1)d$ in number
request message at most $(p-1)d$ in number
no at most $(p-1)d$ in number
yes at most $(p-1)d$ in number
abort at most $(p-2)d$ in number
succeed at most d
Total: $5(p-1)d$ messages

The Best Case

In the best case, there will be no aborts; therefore, a master guard will be successful in $4d$ messages. A slave guard will require four messages for successful execution of a handshake.

5. Correctness of the algorithm

In this section, we prove that the algorithm shown in Figure 9.2 is correct. The correctness of the algorithm is shown in two parts. We show that the algorithm is safe, that is it can ask a process to participate in at most one handshake. We also show that the algorithm is live, that is if one or more handshakes are enabled, the system will execute some handshake.

5.1. Safety Property

Theorem 9.1: Each process can be asked to participate in at most one handshake.

Proof: A process commits for a handshake only if it has sent yes in response for the handshake or it is master for that handshake and has sent request messages. Since a process can have at most one outstanding yes message if it has not sent out any request message and none if it has sent, a process cannot commit for two handshakes. Q.E.D.

5.2. Liveness Property

Theorem 9.2: If one or more handshakes are eligible then the system will execute a handshake.

Proof: Consider the master of the handshake who has waited for the longest time. When this master sends out the request message, if all processes respond with *yes*, the handshake can be executed. Since the handshake is eligible and has the highest priority, no process can send *no* for the handshake. The only other option for them is to delay their response.

We define the delay graph D as a directed graph D = (V, E) where V is the set of all the processes. There is a directed edge from process v_1 to v_2 if there exists a process that has delayed v_1 in favor of v_2. This implies that the priority of v_1 is greater than v_2 because we delay the older process. Global causality implies that the graph is acyclic. We traverse the path of processes in the delay graph. Since the delay graph is acyclic, we will reach a node which has no outgoing edge. This process being youngest will receive answer from all the processes and therefore can send *success/abort* message to all its processes in its set which then can reply to their delayed masters. If the decision was *abort* then the path delay graph has less number of edges and this particular handshake will not be explored again. If the decision was *success*, a handshake is executed. Q.E.D.

For example, consider the example of distributed players. Let Joe be the master of tennis, Mary of chess, Bob of poker and Jack of bridge. Let the last time each game was played be as follows:

Tennis	12
Chess	15
Poker	14
Bridge	18

Assume that tennis is eligible because all participating players are willing to play it. Assume the Bob and Mary are willing to play poker but Jack is not. Consider the following event sequence:

(1) Bob sends a request to Mary for poker who responds yes as she has not committed to any other game.

(2) Joe sends request for tennis to Mary who delays the response to this message.

(3) Bob sends request for poker to Jack who responds with a no message.

(4) Bob sends abort for poker to Mary, who now can respond to the delayed request of Joe.

(5) Joe can now send success message to Mary, who then can execute the handshake.

5.3. Effective Implementation

Theorem 9.3: The algorithm satisfy all the six criteria of effective implementation as proposed by [Buckley 83] and extended by [Back 84, Ramesh 87]. The six criteria are as follows.

(1) The number of processes that are involved in the selection of a guard should be minimum.

(2) The amount of system information that each of these processes should be low.

(3) When a handshake is ready then it will be selected within a finite time.

(4) The number of messages exchanged for making a selection by any process is small.

(5) The time it takes for a process to determine whether it can establish com-

munication with some other process should be bounded.

(6) *If a process has a guarded command that is infinitely often enabled, then it should eventually succeed.*

Proof: (1) and (2) are obvious from the algorithm. (3), (4) and (5) follows from Theorem 9.2 and the message complexity analysis. (6) follows from our fairness conditions. Q.E.D.

6. Efficiency Considerations of the Algorithm

In the above algorithm, we did not discuss how we chose masters for each handshake. The efficiency of the algorithm is dependent on this choice. We discuss some desirable requirements for the choice and strategies to assign masters based on the requirements.

6.1. Minimum Maximum load of any node

The algorithm, as presented above, is unfair with respect to the master of the handshake who may have to deal with more messages than other participating processes. To prevent any machine from getting overloaded, we may choose masters such that the maximum load on any machine is minimized. The problem can be stated formally as follows: Let M and H represent the set of machines and the set of handshakes respectively. Let the degree of a handshake h be the number of machines which participate in it. Our problem is to find an assignment of master for hanshakes, $f:H \rightarrow M$, such that the maximum load on any machine is minimized. The load of a machine is defined as the sum of degrees of all handshakes for which it acts as a master. For example, consider the example in Figure 9.1. The degree of various handshakes is as follows:

Chess: 2
Tennis: 2
Poker: 3
Bridge: 4

A possible master assignment is as follows:
Chess, Poker: Mary
Tennis: Joe
Bridge: Jack

The maximum load in this assignment is on Mary who has to the load of Chess (2) and Poker (3). If Bob is assigned as the master of Poker, then Jack will have the maximum load of Bridge (4).

Theorem 9.4: Let there be m machines and n handshakes. There exists an algorithm with $O(\log(mn)m^2n^2)$ time to find the master assignment f:H->M, such that the maximum load on any machine is minimized.

Proof: We consider a related problem which seeks the assignment of masters such that the maximum load on any machine is less than K. Let the total load (the sum of degree of handshakes) be S.

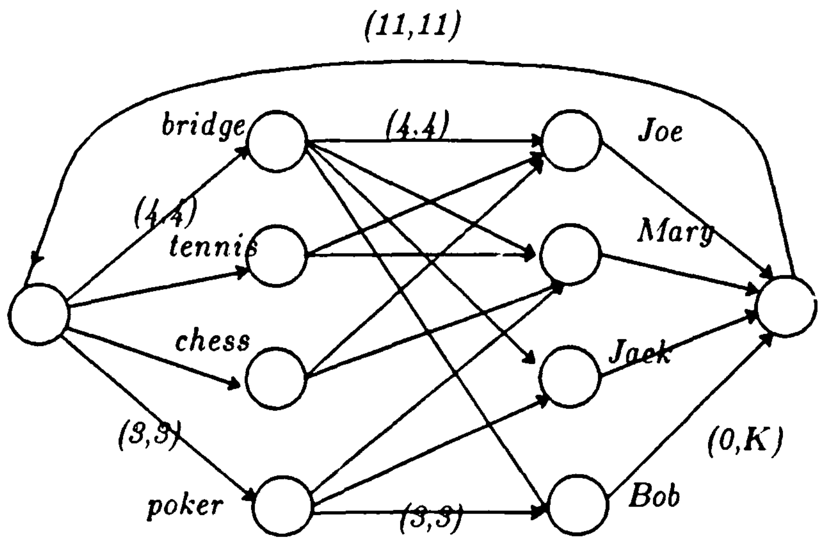


Figure 9.3: Minimizing the Maximum Load

This problem can be solved as feasible circulation in a network with upper as well as lower bounds on the capacity of each edge. We add a pseudo source s and a pseudo sink t with the following bounds:

$$l(s,h_i)=u(s,h_i)=l(h_i,m_j)=u(h_i,m_j)=degree(h_i)$$

$$l(m_i,t)=0; u(m_i,t)=K, l(t,s)=S, u(t,s)=S \quad \forall h_i \in H,\ m_j \in M$$

Figure 9.3 shows the assignments to various edges. Using Out-of-Kilter method[Lawler 76], this problem can be solved in $O(m^2n^2)$ where m is the number of machines and n is the number of handshakes. Using the solution to decision problem, we can solve the minimization problem in $O(log(mn))$ time using a binary search. Thus, the problem of finding master assignment such that the maximum load on any machine is minimized can be solved in $O(log(mn)m^2n^2)$.

6.2. Minimum Total Number of Messages

An alternative optimization criterion could be the minimization of the number of messages required in the overall system. As shown for the calculation of the message complexity of the algorithm, the number of aborts are minimum if the number of processes acting as master is minimum. This is easy to see intuitively. If there are more processes that can act as master, there are greater chances that these processes will attempt a handshake which requires a common process and therefore some of them will be aborted. In the limiting case, (that is if we were allowed to have just one global master), there will be no aborts. This extreme, however, violates our condition on effective implementation which requires that the number of processes involved in deciding if a handshake should be executed must be minimum. Thus, our aim is to minimize the number of masters with the condition that the master must be one of the participants for the handshake. Unfortunately this problem is NP-complete as shown by Theorem 9.5.

Theorem 9.5: The master assignment problem such that the number of masters is miniminzed is NP-complete even for the case where each handshake is shared by exactly two processes.

Proof: We reduce the vertex cover problem known to be NP-complete to master assignment problem by treating vertices as processes and undirected edges as handshakes between these processes. The vertex cover problem is as follows:

Instance: Graph $G=(V,E)$, positive integer $K \leq |V|$.

Question: Is there a vertex cover of size K or less for G, i.e. a subset $V' \subseteq V$ with $|V'| \leq K$ such that for each edge $\{u,v\} \in E$ at least one of u and v belongs to V'.

We reduce each edge $\{u,v\}$ to a handshake between machine u and v. If this handshake is assigned to u then we include u in V' and vice-versa. Q.E.D.

Since the number of processes may not be very large and the computation of master is done only once, it may still be feasible to compute the optimal master assignment. However, since the number of processes may not be very large and the computation of master is done only once, it may still be feasible to compute the optimal master assignment.

7. Conclusions

We have proposed an efficient implementation of the shared events in a distributed environment. Our solution can be used to execute STOCS machines (and therefore, also Petri nets). It is also applicable for implementation of the generalized CSP I/O command. Our algorithm is conceptually simpler and more efficient than existing algorithms.

CHAPTER 10

Conclusions

1. Summary of the Work

In this report, we have tackled the problem of formal specification and analysis of message based asynchronous concurrent systems. We have defined a new model of concurrent computation called the Synchronous Token based Communicating State (STOCS) Model. The STOCS model combines the advantages of net-theoretic and algebraic approaches for the study of concurrent systems. It is amenable to net-theoretic analysis because the reachability problem in a Petri net is reducible to that in a a STOCS machine and vice-versa. It is easier to use than Petri nets as it supports modularity in specification and analysis. For example, we have shown that analysis of safety properties can avoid searching global state space by considering only the relevant modules. To show that the model also supports algebraic specification, we prove that STOCS machines can be characterized by *concurrent regular expressions.* Concurrent regular expressions extend classical regular expressions with three operators - interleaving, alpha closure and synchronous composition. As an application of this result, we provide an algebraic characterization of Petri net languages.

Based on the STOCS model, we propose two new constructs, ***handshake*** and ***unit***, to support concurrent computation. The handshake construct is a generalized remote procedure call for multiple parties. The unit expression is a generalized path expression which provides conditional synchronization by restricting the possible sequence of calls to handshakes. Any program that has its communication aspects specified using these constructs can be analyzed for logical correctness of its communication. We have developed a fair and efficient algorithm for execution of multi-process shared events required for implementation of our constructs. Our implementation extends "C" for concurrent programming and the current version runs on Unix 4.3 BSD. We conclude that the STOCS model is a good starting point for modeling asynchronous concurrent systems based on synchronous communication.

2. Future Work

The novelty and simplicity of the STOCS model has opened up a large number of interesting issues in specification and analysis of concurrent systems. We now discuss some open problems in each of the important aspect of using the STOCS model.

2.1. Specification

• Reduction of non-determinism: A non-deterministic finite state machine can always be converted to a deterministic finite state machine. This fact leads to many advantages, as it might be easier to specify a system using non-deterministic finite state machine but easier to simulate a deterministic finite state machine. Along similar lines, it is desirable to convert a non-deterministic STOCS machine to a deterministic STOCS machine if possible. This may not always be possible as we do not know whether the family of languages accepted by DSTOCS machines is the same as that accepted by STOCS machines. More research is required for algorithms to convert a STOCS machine to a DSTOCS machine.

• Canonical Representation of a STOCS Machine: A deterministic finite state machine can always be minimized with respect to its number of states. Besides saving in the number of states, this has the advantage of providing a canonical representation of a finite state machine. Thus to check whether two finite state machines are identical, we need only convert them to their canonical forms. In an analogous fashion, it is desirable to have a canonical representation of a STOCS machine which can be used to check equivalence between two STOCS machines.

• Language Preserving Transformations on STOCS Machines: A finite state machine can be optimized for its number of states by the minimization algorithm. Similarly, it is desirable to have language preserving transformations on STOCS machines which may minimize the number of places, minimize the number of units, minimize the number of *-places, minimize the non-determinism or maximize the concurrency possible in the system.

• Modeling of General Linear Constraints: In section 3.4, we gave examples of units that modeled some simple linear constraints on the number of occurrences of symbols in a string, such as $n_a=2n_b$. We conjecture that any linear constraint with rational coefficients can be modeled by a unit. The conjecture is true if a set of strings which satisfy a linear constraint also satisfies the property that there exists a constant M, such that any string longer than M can be written as interleaving of strings smaller than M. A constructive proof for the truth of the conjecture will provide a technique to synthesize a STOCS machine from a set of linear constraints.

• Hierarchical Modeling and Analysis: Hierarchical modeling reduces the complexity of the system by providing abstraction. If a system that is expressed hierarchically can also be analyzed hierarchically, then a substantial saving of computational effort may be possible during its analysis. In STOCS model, an internal procedure can be modeled by a transition. More research is required to make the model more amenable to hierarchical specification.

2.2. Relationship with Petri nets

• Polynomial reduction of reachability in STOCS to Petri Nets: A free labeled STOCS can be directly converted to Petri Nets in linear time using Lemma 3. Similarly using Lemma 2 reachability problem in a Petri net can be converted reachability in STOCS using linear time. However, we do not know of any *method to transform reachability* in a general STOCS to a Petri net in polynomial time.

• Exact Relationship with Petri Net languages: From Theorem 5.3, we know that concurrent regular languages are contained in Petri net languages. The question that remains to be answered is: Is the inclusion proper? In other words, are there Petri net languages which are not concurrent regular ? Our current belief is that this is the case. Our belief is based on the fact that Petri net languages are closed under concatenation and union, and this does not seem plausible for concurrent languages. In particular, we have not been able to construct STOCS machine that accepts $(ab)@.(bc)@$.

• Minimum Number of Connected *-places: *-places are the only sources of unboundedness and therefore it is desirable to minimize the number of connected *-places. By our construction, each unit can have at most one *-place. Thus the problem is reduced to decomposing a Petri net into units such that at most K of them have connected *-places. A unit assigned the number K has a connected *-place iff there exists a transition t such that there exist a place assigned the color K as input of the transition, or output of the transition but not both. Rephrasing the above, we get the following problem:

Instance: An ordinary Petri Net N= (P,T,I,O) and $0 \leq K < |P|$

Question: Is there a function f:P->{1,2,..,M} such that $f(p_1) \neq f(p_2)$ whenever $\{p_1, p_2\} \in I(T) \vee (p_1, p_2) \in O(t)$ for some $t \in T$, and for all places p_1 such that $f(p_1) > K$, if $p_1 \in I(t) \cup O(t)$ then $\exists p_2 : f(p_2) = f(p_1) \wedge p_2 \in I(t) \cup O(t)$.

2.3. Algebraic Representation of a STOCS Machine

• Canonical Representation of CRE's: To check the equivalence of two concurrent regular expressions, it is important to have a canonical representation of concurrent regular expressions.

• Optimization of CRE's: It is desirable to reduce the number of []'s or α's in a given expression. This problem may be quite hard, as a similar problem for regular expressions (star-height problem) is still open.

2.4. The Language of a STOCS Machine

• Family of Languages of k-STOCS: A k-STOCS is defined as a STOCS with at most k units. We conjecture that the family of languages accepted by a (k-1)-STOCS is properly contained in k-STOCS. Chapter 5 show that 1-STOCS is properly contained in 2-STOCS. In Garg 88, we have shown that the conjecture is true for FLSTOCS.

• Closure Properties of STOCS Languages: STOCS languages are clearly closed under synchronous composition. As Petri nets are not closed under Kleene Closure, we also know that concurrent regular languages are not either. It is still an open problem whether concurrent regular languages are closed under $+$, $.$, α and $||$.

2.5. Modeling of Uncontrollable Events

• Efficient Construction of URE from UFSM: In this chapter, we provided an efficient construction of UFSM's from URE's. Our construction of URE from a UFSM, however, requires calculation of regular expression for minimal acceptance set the UFSM. As a result, the final expression has a single $\oplus$ but may be very long. We do not know of any method of exploiting $\oplus$ operator more efficiently so that a smaller expression is calculated from a given machine.

• Axiomatic Proof System: We can check the equivalence of two expressions (machines) if they could be converted to a canonical representation. An alternative method is to provide sound and complete axioms and rules of inference such that any relationship between two expressions can be proved.

2.6. Analysis of STOCS Machines

• Analysis of General Regular Topology: In chapter 7, we saw how machines connected in star, broadcast or ring topology can be analyzed. Some of the other interesting topologies are regular topologies such as hyper-cube in which each processor has three neighbors. An interesting task for investigation is the generalization of these techniques for identical processes connected in any arbitrary topology.

• Reachability in k-STOCS: For symbolic reachability, matrix equations led us to necessary but not sufficient conditions for reachability. We are investigating efficient techniques which gives conditions both necessary and sufficient for reachability in a restricted class of STOCS model: STOCS machines with at most k units.

• Stopping Rule for Induction: For application of the induction technique, we need to find a k such that the system with k processes is equivalent to a system with $k+1$ processes. It was easy in our examples where k had small values(1 and 2). There needs to be a more general algorithm for selecting k.

• Performance Analysis: Timed Petri Nets and Stochastic Petri nets have been used extensively for performance analysis of diverse kind of systems. It is easy to extend definitions of the STOCS model to simulate Timed or Stochastic Petri nets, but it remains to be seen if the modularity in STOCS model is also beneficial in performance analysis of concurrent systems.

2.7. Incorporation of the STOCS model in a Programming Language

• Naming: The current design uses explicit naming of processes in the spirit of CSP. As for CSP, this may prove restrictive and use of port names may be preferred. We have chosen to keep the initial prototype simple and the future design may include port names.

• Extension to Asynchronous Communication primitives: The current process allows only synchronous communication primitives. Asynchronous message passing can be specified using an extra buffer process. We chose to keep synchronous primitives only, as reasoning with asynchronous processes is error prone and cumbersome.

• Dynamic Process Structure: The current design also restricts the process structure to be static. This implies that unbounded process activation and recursive process activation is not possible. This restriction is a direct consequence of our aim of keeping the construct analyzable.

• Exception Handling, Fairness: The current design assumes an error free reliable message service. It also does not address the issues of process failures, reliability, exception handling and security. Similarly specification of priority and issues arising due to fairness concerns are not considered here. The notion of time is also missing in the current design.

In conclusion, this dissertation is a first step towards a model that combines advantages of net-theory and algebraic theory of concurrent systems.

References

[Ackerman 82]W.B. Ackerman, "Data flow languages", Computer 15,2, pp 15-23, 1982.

[Aggarwal 87]S.Aggarwal, D. Barbara, K.Z. Meth, "SPANNER: A Tool for Specification, Analysis, and Evaluation of Protocols", IEEE Transactions on Software Engineering, Vol 13, 12 December 1987, pp 1218-1237.

[Aggarwal 84]S.Aggarwal, R.P.Kurshan, "Automated Implementation from Formal Specification", Protocol Specification, Testing, and Verification, IV, North Holland 1984.

[Alford 85] M. Alford et. al., "Distributed Systems - Methods and Tools for Specification", Springer Verlag, LNCS 190, 1985.

[Ada 83] Reference Manual for the Ada Programming Language, United States DoD, Washington, ANSI/MIL-STD-1815A-1983, 1983.

[Anderson 88]D.P.Anderson, "Automated Protocol Implementation with RTAG", IEEE Transactions on Software Engineering, Vol 14, 3 March 1988, pp 291-300.

[Apt 80] K.Apt, N.Francez, W.de Roever,"A Proof System for Communicating Sequential Processes", ACM TOPLAS 2,3(July 1980)

[Andrews 82] G.R.Andrews, "The Distributed Programming Language SR - Mechanisms, design and implementation", Software Practice and Experience 12, 8 ,Aug 1982, pp 719-754

[Andrews 82] G.R.Andrews, F.B.Schneider, "Concepts and Notations for Concurrent Programming", Computing Surveys, Vol. 15, No. 1, March 1983, pp 3-43.

[Back 84] R.J.R. Back, P. Eklund, and R. Kurki-Suonia, "A fair and efficient implementation of CSP with outpur guards", Tech. Report No. 38, Abo Akademi, Finland, 1984.

[Backus 78] J. Backus, "Can Programming be Liberated from the von Neumann Style", Comm. ACM, 21, 8, pp 613-641, 1978.

[Bagrodia 86] Rajive Bagrodia, "A Distributed Algorithm to Implement the Generalized alternative command of CSP", Proc. of International Conference on Distributed Computing Systems (ICDCS), pp 422-427, 1986.

[Bernstein 80] A.J. Bernstein, "Output Guards and Non-Determinism in CSP", ACM Toplas, 2(2), April 1980, pp 234-238.

[Billington 88] J.Billington,G.R.Wheeler, M.C.Wilbur-Ham, "PROTEAN: A High-Level Petri Net Tool for the Specification and Verification of Communication Protocols", IEEE Transactions on Software Engineering, Vol 14, 3 March 1988, pp 301-316.

[Blumer 86] T.P.Blumer and D.P.Sidhu, "Mechanical Verification of Automatic Implementation of Communication Protocols", IEEE Trans. on Softw. Engg., Vol 12, 8 August 1986, pp 827-843.

[Bochmann 80] G.v.Bochmann, P. Merlin, "On the construction of communication protocols", in Proc. Inter. Conference on Computer Communication, 1980.

[Brinch Hansen 75]

P. Brinch Hansen, "The Programming Language Concurrent Pascal", IEEE Trans. Softw. Engg., SE-1,2, pp 151-164, June 1975.

[Brinch Hansen 78]

P. Brinch Hansen, "Distributed Processes: A concurrent Programming Concept", Comm. ACM 21, 11, Nov 1978, pp 934-941.

[Buckley 83] G.N.Buckley, A.Silberschatz, "An Effective Implementation for the Generalized Input-Output Construct of CSP", ACM transactions on programming languages and systems (TOPLAS), April 1983.

[Campbell 74]R.H.Campbell, A.N.Habermann, "The Specification of Process Synchronization by Path Expressions", Lecture Notes in Computer Science, vol 16, Springer Verlag, New York 1974, pp 89-102.

[Campbell 79]R.H.Campbell, R.B.Kolstad, "Path Expressions in Pascal", Proc. 4th International Conference on Software Engineering, Munich, IEEE New York, 1979, pp 212-219.

[Cerf 72] V.Cerf, "Multiprocessors, Semaphores, and a Graph model of Computation," Ph.D. dissertation, Computer Science Department, University of California, Los Angeles, California, April 1972.

[Chandy 85] K.M.Chandy, "Concurrent Programming for the Masses", Proc. 4h Principles of Distributed Computing, 1985.

[Charlesworth 87]
A. Charlesworth, "The Multiway Rendezvous", ACM Trans. on Programming Languages and Systems, Vol 9, No.2, July 1987, pp 350-366.

[Clarke 86] E.M.Clarke, O. Grumberg and M.C.Browne, Reasoning about Networks with many Identical Finite-State Processes, proc. of Principles of Distributed Computation, 1986.

[Courtois 71] P.J.Courtois, F.Heymans, D.L.Parnas, "Concurrent Control with readers and writers", Comm. ACM, 10, pp. 667-68, Oct 1971.

[Dijkstra 85] Invariance and Non-determinacy, in Mathematical Logic and Programming Languages, C.A.R. Hoare and J.C. Shepherdson, Eds. Prentice-Hall, 1985, pp 157-163.

[Dong 83] S.T.Dong, "The Modeling, Analysis, and Synthesis of Communication Protocols", Ph.D. Dissertation, UC Berkeley, 1983.

[Eswaran 76] K.P.Eswaran, J.N.Gray, et.al.,"The Notion of Consistency and Predicate Locks in a Database System", Comm. ACM, 18(11), Nov

1976, pp 624-633.

[Feldman 79] J.A.Feldman, "High Level Programming for Distributed Computing", Comm. ACM 22, 6, June 1979, pp 353-368.

[Filman 84] R.E.Filman and D.P. Friedman, "Coordinated Computing, Tools and Techniques for Distributed Software", McGraw-Hill, 1984.

[Forman 86] I.R.Forman, "On the Design of Large Distributed Systems", Proc. International Conference on Computer Languages, 1986.

[Francez 83] N.Francez, B.Hailpern, "Script: A Communication Abstraction Mechanism", Proc. of 2nd Symposium on Principles of Distributed Computing, 1983.

[Garey 79] M.R. Garey, D.S. Johnson," Computers and Intractability, A Guide to the Theory of NP-Completeness", W.H. Freeman and Company, 1979.

[Garg 88a] V.K.Garg, "Specification and Analysis of Concurrent Systems Using STOCS model", Proc. of Computer Networking Symposium, Washington D.C. 1988.

[Garg 88b] V.K.Garg, "Analysis of Distributed Systems with many Processes", Proc. International Conference on Distributed Computing Systems, San Jose, 1988.

[Garg 88c] V.K.Garg, C.V. Ramamoorthy, "High Level Communication Primitives for Concurrent Systems" Proc. International Conference on Computer Languages, Miami, 1988.

[Gehani 84] N.H. Gehani, W.D. Roome, "Concurrent C", Memorandum, Bell Labs, Murray Hill, 1984.

[Genrich 80] H.J. Genrich, K. Lautenbach and P.S. Thiagarajan, "An overview of Net Theory", Proc. Advanced Course on General Net Theory of Processes and Systems, LNCS 1980.

[Gerhart 80] S.L.Gerhart, et al., "An Overview of Affirm: A Specification and Verification System", Proc. IFIP 80, pp 343-348, Australia, October 1980.

[Hack 75] M.Hack, "Decision Problems for Petri Nets and Vector Addition Systems", Tech. Memo 59, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts (March 1975).

[Hack 75] M. Hack, "Petri Net languages", Computation Structures Group Memo 124, Project Mac, Massachusetts Institute of Technology. June 1975.

[Hewitt 79] C. Hewitt, G. Attardi, H. Lieberman, "Specifying and Proving Properties of Guardians for Distributed Systems", LNCS 70, 1979.

[Hoare 84] C.A.R.Hoare, S.D.Brookes, A.W. Roscoe, "A Theory of Communicating Sequential Processes", Journal ACM, Vol 13, No 3, pp 560-599, July 1984.

[Hoare 85] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall. Inc., Englewood Cliffs, New Jersey 1985.

[Hopcroft 79] J.Hopcroft and J.Ullman, "Introduction to Automata Theory, Languages, and Computation", Addison-Wesley Pub. Co., Reading.

[INMOS 84] Occam Programming Manual, Prentice-Hall International, pp. 100, 1984.

[Kahn 77] G. Kahn, D.B.MacQueen, "Coroutines and networks of parallel processes", Information Processing 77, North Holland, 1977.

[Kanellakis 85]P.C. Kanellakis, S.A. Smolka, "On the Analysis of Cooperation and Antagonism in Networks of Communicating Processes", Proc. Fourth ACM Symposium on Principle of Distributed Computing, Canada, 1985, pp 23 - 38.

[Karp 68] R.Karp, and R.Miller, "Parallel Program Schemata", RC-2053, IBM T.J. Watson Research Center, Yorktown Heights, New York (April

1968).

[Kurshan 85] R.P.Kurshan, "Modeling Concurrent Processes", Proc. of Symposia in Applied Mathematics, 1985.

[Lamport 78] L.Lamport, "Time, Clocks and Ordering of Events in a Distributed System" Comm. ACM, 21(7), July 1978, pp 558-565.

[Lamport 84] L.Lamport, F.B. Schneider, "The 'Hoare Logic' of CSP and All That", ACM TOPLAS 6,2(April 1984).

[Lampson 81] Lampson et. al., "Distributed Systems - Architecture and Implementation", Springer Verlag, LNCS Vol 105, 1981.

[Lauer 75] P.Lauer, R. Campbell, "Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes" Acta Informatica, Vol 5, Number 4 1975, pp 297-332. pp 441-460, August 1984.

[Lauer 79] P.E. Lauer, P.R. Torrigiani, M.W.Shields, "COSY: A System Specification Language Based on Paths and Processes", Acta Informatica 12, pp 109-158, 1979.

[Lawler 76] E. Lawler, "Combinatorial Optimization - Networks and Matroids", Holt, Rinehart and Winston, 1976.

[Lipton 76] R. Lipton, "The Reachability Problem Requires Exponential Space", Research Report 62, Department of Computer Science, Yale University, Connecticut, 1976.

[Lee 87] I. Lee, S.B. Davidson, "Generalized I/O with Timing Constraints", Proc. of International Conference on Distributed Computing Systems (ICDCS), pp 316-323, 1987.

[Levin 81] G.Levin, D.Gries, "Proof Techniques for Communicating Sequential Processes", Acta Informatica 15, 1981, pp281-302.

[Li 85] W.Li, P.E.Lauer, "Using the Structural Operational Approach to Express True Concurrency", Formal Models in Programming, Elsevier Science Publishers, (North-Holland), pp 373-397, 1985.

[Liskov 84] B. Liskov,"The Argus Language and System", Proc. Advanced Course on Distributed Systems - Methods and Tools for Specification, TU Munchen, Apr. 1984.

[Mayr 84] E.W.Mayr, "An Algorithm for the General Petri Net Reachability Problem", SIAM Journal of Comput., Vol. 13, No.3 pp 441-460, August 1984.

[Milner 80] A Calculus of Communicating Systems, Lecture Notes in Computer Science, Vol 92, Springer-Verlag 1980.

[Milne 85] G.J.Milne,"CIRCAL and the Representation of Communication, Concurrency and Time," ACM TOPLAS, 7(2), pp 270-298, April 1985.

[Misra 81] J.Misra, K.M.Chandy, "Proofs of Networks of Processes", IEEE Trans. on Softw. Engg. SE-7,4(July 1981) pp 417-426.

[Misra 82] J.Misra, K.M.Chandy, "Proving Safety and Liveness of Communicating Processes with Examples", Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Aug 1982, pp 201-208.

[Mitchell 79] J.G.Mitchell, W.Maybury, R.Sweet, "Mesa Language Manual, version 5.0" Rep. CSL-79-3, Xerox Palo Alto Research Center, April 1979.

[Murata 84] T. Murata, "Modeling and Analysis of Concurrent Systems", in book Handbook of Software Engineering, ed. C.R.Vick and C.V.Ramamoorthy, Publ.Van Nostrand Reinhold, pp 39-63, 1984.

[Needham 82]R.M. Needham,A.J.Herbert, "The Cambridge Distributed Computing System", Publ. Addison-Wesley Publishing Company, 1984.

[Nivat 82] M. Nivat, "Behaviors of Processes and Synchronized Systems of Processes", Theoretical Foundations of Programming Methodology, Reidel Dodrecht, pp 473-550, 1982.

[Owicki 82] S.Owicki, L.Lamport,"Proving Liveness Properties of Concurrent Programs", ACM TOPLAS 4,3 (July 1982).

[Peterson 81] J. Peterson, Petri-Net Theory and Modeling of Systems, Prentice Hall, Inc., Englewood Cliffs, New Jersey 1981.

[Petri 62] C.Petri,"Kommunkation mit Automaten,"Ph.D. dissertation, University of Bonn, Bonn, West Germany, 1962.

[Pratt 82] V.R. Pratt, "On the Composition of Processes", Proc. 9th POPL, Albuquerque, New Mexico 1982.

[Queille 82] J.P. Queille, J.Sifakis, "Specification and verification of concurrent systems in CESAR" Inter. Symp. on Programming LNCS 137 pp337-350, 1982.

[Ramesh 86] S.Ramesh, "Programming with Shared Actions: A methodology for developing Distributed Programs", Ph.D. Dissertation, IIT Bombay, India, June, 1986.

[Ramesh 87] S.Ramesh, "An Efficient Implementation of CSP with Output Guards", Proc. of International Conference on Distributed Computing, 1987.

[Raynal 88] M. Raynal, "Distributed Algorithms and Protocols", John Wiley and Sons, 1988.

[Reif 84] J. H. Reif, P.G. Spirakis, "Real-Time Synchronization of Interprocess Communications", ACM transactions on programming languages and systems (TOPLAS), pp 215-238, April 1984.

[Reisig 85] W. Reisig, Petri Nets, An Introduction, lecture notes in Computer Science, Springer-Verlag, 1985.

[Silberschatz 79]
A. Silberschatz, "Communication and Synchronization in Distributed Systems", IEEE Transactions Software Eng.-5,6 Nov. 1979, pp 542-546.

[Snepscheut 81]
J.L.A. Van de Snepscheut, "Synchronous Communication between asynchronous components", Information Processing Letters, 13, 3 Dec. 1981, pp 127-130.

[Stark 87] E. Stark, "Concurrent Transition System Semantics of Process Networks", Proc. POPL, pp 199-210, 1987.

[Steenstrup 83]Port Automata and the algebra of concurrent processes, JCSS, 27(1), pp 29-50, 1983.

[Soundarajan 81]
N. Soundarajan, "Axiomatic Semantics of Communicating Sequential Processes", Tech. Report, Department of Computer and Information Sciences, Ohio State University, 1981.

[Sunshine 78] C.A.Sunshine,"Survey of Protocol Definition and Verification Techniques", Proc. of the Computer Network Protocols Symposium, Liege, Belgium, 1978.

[Suzuki 83] I.Suzuki and T.Murata, "A Method for Stepwise Refinement and Abstraction of Petri-nets", J. of Comp. and Syst. Sci., Vol 27, 7 pp 51-76, August 1983.

[Winskel 82] G. Winskel, "Event Structure Semantics for CCS and Related Languages" Proc. 9th ICALP, LNCS 140, Springer Verlag, pp 561-576, 1982.

[Zave 85] P.Zave, "A Distributed Alternative to Finite-State-Machine Specifications", ACM Transactions on Programming Languages and Systems Vol 7, No 1, January 1985, pp 10-36.

```
/*                          Appendix A                          */
/* a program here refers to communication component of a
single process. Each process is supposed to have a computational
component written in C and a communication component written in
handshake and units. Handshakes which are common to multiple
processes are replicated. It is assumed that the first process
mentioned in the handshake is responsible for executing the
body. */

program     :
            YPROCESS YID ';'
                    configinfo
                    unit
                    handshake_list
            ;


/*
**********************************************************
      Definition of configuration
**********************************************************
*/

configinfo:
              YCONFIGURATION hostlist YEND YCONFIGURATION ';'
              |
              ;

hostlist: hostlist YID ':' YID ';'
              |
              YID ':' YID ';'

              ;
/*
**********************************************************
      Definition of a unit
**********************************************************
*/
unit          :
            YUNIT
            communitname
            ';'
            unitinfo
            state list
            YEND YUNIT ';'
            ;


unitinfo:
            YSTATE YNUMBER ';'
            stateval
            marking
            ;

stateval:
            YCONST '[' vallist ']' ';'
            ;

vallist:
            vallist ',' valitem
            |
            valitem
            ;
```

```
valitem: YID ':' YNUMBER
          ;

marking:
          YMARKING '[' mlist ']' ';'
          ;

mlist:
          mlist ',' mitem
          |
          mitem
          ;

mitem: YID ':' YNUMBER
                  |
          YID ':' '*'
                  ;

/*
*********************************************************
          Definition of state transitions
*********************************************************
*/
state_list:
          statedes
          |
          state_list statedes
          ;

statedes:
          stateinfo trans_list
          ;

stateinfo:
          statename
          action ';'
          ;


trans_list:
          transdes
          |
          trans_list transdes
          ;

transdes:
          '>' transname
           statename action ';'
          ;

action: YACTION
          |
          ;
communitname: YID
          ;

statename: YID
          ;

transname: YID
          ;
```

```
/*
***************************************************************
           Definition of handshakes
***************************************************************
*/
handshake_list: handshake
           |
           handshake_list handshake
           ;

handshake: handshakeheader
               hbody
               ;

handshakeheader: enableinfo YHANDSHAKE transname ';'
           proc_decl
           ;

enableinfo:
           |
           YENABLED
           ;

proc_decl:procedure
           |
           proc_decl procedure
           ;

procedure: YID '.' YID

           '(' arglist ')' ';'
           ;

arglist:arg arglist
           |
           ;

arg:       qual YID ':' YID ';'
           ;

qual:      YVAR
           |
           ;

hbody: YACTION
           |
           ;
%%
#include "lex.yy.c"
```